Interoperabilidade na prática: Aplicativo Delphi acessando um servidor WCF .NET

Clube Delphi

Muito além do banco de dados
Conheça o Rapid SQL XE5

Atualização automática de arquivos
Solução completa em Delphi

ISSN 1517990-7

Edição 157 :: R$ 14,90

DEVMEDIA

ENGENHARIA REVERSA

Crie classes a partir de um banco de dados

Inovação em modelagem de banco de dados
Explore os recursos do TMS Data Modeler

Refatoração no Delphi XE5
Melhore seu código com a ajuda do IDE

CURSOS ONLINE

Clube Delphi

A Revista Clube Delphi oferece aos seus assinantes uma série de Cursos Online de alto padrão de qualidade.

CONHEÇA ALGUNS DOS CURSOS

- Curso de Multicamadas com Delphi e DataSnap
- Delphi para Iniciantes
- Criando componente Boleto em Delphi
- Loja Virtual em Delphi Prism

Para mais informações :

www.devmedia.com.br/cursos/delphi

(21) 3382-5038

http://www.devmedia.com.br

DEVMEDIA

Clube Delphi

157ª Edição 2014 ISSN 1517990-7 Impresso no Brasil

Editor Geral

Paulo Quicoli (pauloquicoli@gmail.com)

Equipe Editorial

Guinther Pauli (guintherpauli@gmail.com)

Fabrício Hissao Kawata (fabricio.kawata@bol.com.br)

Giuliano Scombatti Pinto (giuliano@sygnux.com.br)

Daniel Sobrinho Laporte (daniel.laporte@gmail.com)

Jornalista Responsável

Kaline Dolabella - JP24185

Consultor Técnico

Wesley Yamazack (yamazack@gmail.com)

Capa

Romulo Araujo

Diagramação

Janete Feitosa

Atendimento ao Leitor

A DevMedia conta com um departamento exclusivo para o atendimento ao leitor. Se você tiver algum problema no recebimento do seu exemplar ou precisar de algum esclarecimento sobre assinaturas, exemplares anteriores, endereço de bancas de jornal, entre outros, entre em contato com:

www.devmedia.com.br/central
(21) 3382-5038

Publicidade

Para informações sobre veiculação de anúncio na revista ou no site e para fechar parcerias ou ações específicas de marketing com a DevMedia, entre em contato com:

publicidade@devmedia.com.br

Distribuição

FC Comercial e Distribuidora S.A
Rua Teodoro da Silva, 907
Grajaú - RJ - 206563-900

Fale com o Editor!

É muito importante para a equipe saber o que você está achando da revista: que tipo de artigo você gostaria de ler, que artigo você mais gostou e qual artigo você menos gostou. Fique a vontade para entrar em contato com os editores e dar a sua sugestão!

Se você estiver interessado em publicar um artigo na revista ou no site ClubeDelphi, entre em contato com os editores, informando o título e mini-resumo do tema que você gostaria de publicar:

Paulo Quicoli - Editor da Revista
pauloquicoli@gmail.com

Paulo Quicoli - **Editor Geral**

pauloquicoli@gmail.com
quicoli.wordpress.com
twitter.com/pauloquicoli

MVP

Assine agora e tenha acesso a todo o conteúdo da DevMedia: **www.devmedia.com.br/mvp**

DEVMEDIA

Sumário

Conteúdo sobre Boas Práticas

04 – Engenharia Reversa de Banco de Dados
[*Rodrigo Mourão*]

Artigo no estilo Mentoring

12 – Refatoração no Delphi XE 5
[*Guinther Pauli*]

Artigo no estilo Solução Completa

26 – Atualização de artigos em Delphi
[*Gutierry Antonio*]

Conteúdo sobre Novidades

33 – Rapid SQL XE5: Conheça seu Banco de Dados
[*Fabricio Hissao Kawata*]

Conteúdo sobre Novidades

42 – TMS Data Modeler: Desenvolvendo um projeto de Banco de Dados
[*Fabricio Hissao Kawata*]

Dê seu feedback sobre esta edição!

A ClubeDelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre esta edição, artigo por artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Engenharia Reversa de Banco de Dados

Gere classes de negócios a partir de banco de dados relacionais com um clique

Todo programador começa sua trajetória de aprendizado no mundo de desenvolvimento com o famoso exemplo Hello World, seja qual for a linguagem isso é um fato. Depois que passamos desta fase e decidimos desenvolver nossa primeira aplicação iniciamos pelo modelo tradicional de desenvolvimento estruturado ou procedural como alguns costumam chamar, e nesta estrutura o desenvolvimento da aplicação se inicia pela modelagem do banco de dados. Damos mais ênfase ao projeto e normalização do banco de dados, as tabelas principais e tabelas auxiliares, chaves e relacionamentos. Com o passar do tempo vamos adquirindo mais experiência, aprofundando nosso conhecimento e naturalmente começamos a estudar a orientação a objetos. Neste momento somos impelidos a passar a desenvolver todos os nossos sistemas seguindo este modelo de orientação a objetos e é ai que começam a surgir várias dúvidas. Devo começar do zero? Devo aproveitar minha aplicação atual? Começo a modelar as classes e depois gero o banco? Gero o banco primeiro e depois crio as classes? Enfim, estas e outras dúvidas surgem e em busca das respostas muitas pessoas desanimam e desistem, principalmente quando decidem migrar suas aplicações procedurais para o modelo OO. O trabalho para criar manualmente cada uma das classes de negócio deixa qualquer um desmotivado, pois, por menor que seja uma aplicação a quantidade de tabelas no banco de dados geralmente passa de 100.

O objetivo deste artigo não é responder a todas as perguntas que surgem quando pensamos em desenvolver uma aplicação orientada a objetos, mas independente de estratégia que decidimos adotar, seja começar pela modelagem de depois criar seu banco de dados, seja partir de um banco de dados relacional para gerar seu modelo uma coisa é certa, se decidimos programar utilizando a orientação a objetos vamos precisar de duas coisas: 1) Nossas classes de negócios para que possamos instanciar nossos objetos e dar vida a nossa aplicação e 2) Um banco de dados para persistir nossos objetos. Ora se temos por certo que precisamos das classes e do banco de dados e se começamos nossa trajetória pelo desenvolvimento de sistemas pelo banco de dados, o melhor dos mundos seria poder gerar nossas classes a partir de banco de dados que já temos pronto e em funcionamento e assim pularmos uma etapa chata e demorada que seria criar manualmente cada uma das classes de negócios para as tabelas do nosso banco de dados.

É importante citar que o banco de dados relacional que será alvo da engenharia reversa (ver **BOX 1**) deverá atender a alguns requisitos, com por exemplo, o nome das tabelas deve seguir um padrão, as chaves primárias devem ser únicas, os campos devem seguir também um padrão. Isso é necessário para facilitar a criação do nosso algoritmo que fará a engenharia reversa. Porém sabemos que nem sempre é possível atender a estes requisitos até porque nossos bancos foram criados sem nem mesmo pensar que um dia faríamos uma engenharia reversa nestes bancos.

Nosso algoritmo cuidará tão somente de gerar as classes de negócio, não nos preocuparemos por exemplo com o mecanismo de persistência que é utilizado para persistir classes no banco de dados, justamente porque existem diversas estratégias para isso, você poderia utilizar por exemplo o modelo DAO de persistência (ver **BOX 2**) ou criar um mecanismo de reflexão para mapear os

Fique por Dentro

Este artigo é útil, pois, mostrará como utilizar conceitos simples para resolver problemas complexos. Veremos como implementar um algoritmo para extrair os meta dados de um banco de dados relacional e a partir desta extração gerarmos automaticamente uma classe para cada uma das tabelas deste banco de dados mapeando também cada campo da tabela em um atributo desta classe. Aproveitaremos o conhecimento em RTTI e atributos para "decorar" estas classes com alguns atributos chaves que poderão ser muito úteis caso queiramos criar um mecanismo de mapeamento objeto relacional.

objetos e persisti-los no banco de dados. O objetivo é criar classes que possam ser usadas em qualquer uma das duas situações e mais uma vez repito, você terá liberdade de adaptar o algoritmo a sua realidade.

Embora a persistência não seja o foco do artigo, em nosso algoritmo lançaremos mão do conceito de Custom Attributes do Delphi (ou atributos customizados), que é um conceito presente no Delphi desde a versão 2010 quando a nova estrutura de RTTI (ver **BOX 3**) foi apresentada. Isso se faz necessário primeiro para que possamos aprender onde aplicar na prática os atributos e segundo para deixar nosso algoritmo flexível a tal ponto para atender, como já citado, ambas as estratégias de persistência de dados.

Nota

Caso seu banco de dados não esteja dentro de um padrão de nomenclatura de tabelas e campos não se preocupe, adapte o algoritmo para se adequar a sua realidade e trate as exceções pontualmente.

BOX 1. Engenharia Reversa

A Engenharia Reversa é uma atividade que trabalha com um produto existente (que pode ser um software, uma peça mecânica ou um banco de dados) e tenta entender como este funciona, o que ele faz, quais são suas propriedades, etc. A Engenharia Reversa também é vista como um processo de análise de um sistema para criar uma representação deste mesmo sistema em um nível mais alto de abstração. No caso de engenharia reserva de banco de dados, essa abstração em geral se dá mapeando as propriedades deste banco e gerando seu modelo lógico ou físico para melhor entendimento. Em nosso exemplo faremos a engenharia reserva de um banco de dados com o objetivo de criar a abstração deste banco em um modelo orientado a objetos.

BOX 2. DAO

DAO é o acrônimo de Data Access Object e é um padrão para persistência de dados que permite separar regras de negócio das regras de acesso a banco de dados. Numa aplicação que utilize a arquitetura MVC, por exemplo, todas as funcionalidades de bancos de dados, tais como obter as conexões, mapear objetos para tipos de dados ou executar comandos SQL, devem ser feitas por classes de DAO. De uma maneira geral quando usamos este padrão para cada classe de negócio da nossa aplicação teremos uma classe DAO de persistência.

BOX 3. RTTI

Runtime Type Information (frequentemente abreviado para RTTI) é uma técnica disponível em algumas linguagens de programação e que consiste em manter informação sobre o tipo de dado de um objeto em memória durante o tempo de execução de um programa. Algumas implementações limitam-se a manter somente a árvore de heranças da classe enquanto outras também incluem informações dos métodos do objeto e seus atributos. No Delphi temos uma poderosa estrutura que nos permite em tempo de execução extrair informações detalhadas dos nossos objetos e que nos oferece uma infinidade de possibilidades.

Atributos Customizados

Ao utilizar os operadores "is" e "as" em um código estamos indiretamente utilizando a RTTI do Delphi, pois, estes operadores permitem respectivamente testar o tipo de um objeto e efetuar uma conversão em tempo de execução. O RTTI permite extrair informações mais detalhadas de um objeto, o próprio Delphi faz uso de RTTI em várias partes do IDE, por exemplo, ao clicar em um componente o Object Inspector exibe as propriedades do componente selecionado.

Até a versão 2009 a RTTI mapeava apenas propriedades declaradas no escopo published das classes, porém no Delphi 2010 a RTTI foi totalmente reformulada e agora permite acessar métodos, propriedades e fields em qualquer escopo. Vale lembrar ainda que uma nova API foi criada permitindo assim trabalhar com RTTI de forma mais fácil e transparente, reduzindo bastante a necessidade de codificação, na prática estamos fazendo mais com menos. Agora a maior evolução na RTTI do Delphi 2010 foi sem dúvida nenhuma a introdução dos Atributos Customizados, ou como são conhecidos Custom Attributes.

Os Atributos Customizados são como informações adicionais que podemos inserir em qualquer item do nosso modelo OO, seja nas classes, propriedades, campos ou métodos, informações estas incluídas em tempo de projeto e que podem ser acessadas em tempo de execução pela nova API da RTTI. Os atributos customizados nos permitem estender nosso modelo OO e em geral são muito utilizados na construções de frameworks como por exemplo os de mapeamento objeto relacional. Estes atributos ou informações adicionais não modificam o comportamento dos objetos ou alteram qualquer item no seu modelo, eles apenas acrescentam informações que são usadas em tempo de execução para um determinado fim. Imagine por exemplo que se queira desenvolver um framework para gerar arquivos XML para atender a integração com órgãos públicos. Cada órgão disponibiliza um layout onde os nós do documento XML devem respeitar padrão de nomes, tamanho entre outros detalhes. Pois bem, é possível incluir atributos customizados em cada uma das suas classes e propriedades para adicionar estas informações como nome e tamanho do nó para que, ao mapear o objeto possa-se saber de imediato o nome que será dado ao nó que o representa e seu tamanho conforme especificação.

Um atributo customizado é declarado como uma classe simples que obrigatoriamente deve herdar de TCustomAttribute, além disso uma classe de atributo customizado não pode ser declarada como classe abstrata e não deve conter nenhum método abstrato. A **Listagem 1** mostra um exemplo da declaração de um atributo customizado.

Listagem 1. Exemplo de Atributo Customizado

```
TMeuAtributo = class(TCustomAttribute)
private
  FDescricao: String;
public
  constructor Create(aDescricao: String);
  property Descricao: String read FDescricao write FDescricao;
end;
```

Criando os Atributos

Já que vamos utilizar atributos customizados em nossas classes que serão geradas pela engenharia reversa, nada melhor do que

declarar todos os atributos que usaremos em uma unit. Lembre-se que mesmo não sendo objeto deste artigo a questão do mapeamento objeto relacional, lançaremos mão deste recurso para que nosso framework, que fará o trabalho pesado de converter as tabelas em classes, possa ficar o mais completa possível. Note na **Listagem 2** que criamos uma unit chamada uAtributos e nela declaramos três classes para representar nossos atributos customizados, são elas: TTable, TColumn e TPrimaryKey.

Listagem 2. Unit uAtributos com os Custom Attributes declarados

```
unit uAtributos;

interface

type
 TTable = class(TCustomAttribute)
 private
  FName: String;
 public
  constructor Create(aName: String);
 end;

 TColumn = class(TCustomAttribute)
 private
  FName: String;
  FLength: Integer;
 public
  constructor Create(aName: String; aLength: Integer); overload;
  constructor Create(aName: String); overload;
  property Name: String read FName write FName;
  property Length: Integer read FLength write FLength;
 end;

 TPrimaryKey = class(TCustomAttribute)
 private
  FNameField: String;
 public
  constructor Create(aNameField: String);
  property NameField: String read FNameField write FNameField;
 end;


implementation

constructor TTable.Create(aName: String);
begin
 FName := aName;
end;

constructor TPrimaryKey.Create(aNameField: String);
begin
 FNameField := aNameField;
end;

constructor TColumn.Create(aName: String; aLength: Integer);
begin
 FName := aName;
 FLength := aLength;
end;

constructor TColumn.Create(aName: String);
begin
 FName := aName;
end;

end.
```

Os nomes são bem sugestivos, estes atributos irão adicionar às nossas classes e propriedades informações inerentes as tabelas, campos e chave primárias das entidades do banco de dados que será alvo da engenharia reversa. Com isso, caso você venha desenvolver futuramente uma framework de mapeamento objeto relacional poderá lançar mão destes atributos customizados para gerar os comandos SQL para persistência.

Mas talvez ainda haja a pergunta: "para que eu preciso destes atributos customizados? Eu poderia utilizar o nome da classe para mapear as tabelas e das propriedades para mapear os campos". Na prática sim, mas como é de conhecimento de todo programador Delphi, embora não seja obrigatório é de praxe declarar as classes no Delphi iniciando pela letra T, por exemplo, a classe Cliente é declarada como TCliente e no seu banco de dados não possuímos uma tabela TCliente, mas sim Cliente. Neste caso o atributo customizado TTable faz sua parte adicionando à classe TCliente uma informação que ela deve ser mapeada para uma tabela de nome Cliente, ou ADM_CLIENTE ou seja lá qual for o nome da sua tabela.

Desta forma conseguimos manter o padrão de declaração de classe no Delphi sem que tenhamos dor de cabeça ou façamos gambiarras para mapear a tabela no banco de dados e o melhor sem alterar nosso modelo de classes. Lembre-se, estamos criando um algoritmo para fazer engenharia reversa de um banco de dados que já está em uso e que possui o mais variado padrão de nomenclatura. Note que os construtores que criamos para nossos atributos customizados possuem como parâmetro as propriedades do próprio atributo, isso faz parte da sintaxe de declaração dos atributos customizados. Desta forma quando fizermos uso do atributo em nossa classe o valor passado no momento da declaração será transferido pelo compilador para o construtor do atributo que manterá a informação em memória aguardando para ser extraída via RTTI por quem desejar.

Ainda sobre os construtores, nota-se no atributo TColumn a presença de construtores sobrecarregados (ver **BOX 4**) fizemos isso pois este atributo possui duas propriedades e declarar um construtor com apenas um parâmetro obrigaria a quem fosse utilizá-lo a informar valores para as duas propriedades. Para evitar que isso aconteça nós criamos dois construtores sobrecarregados, um exigindo apenas dados para uma propriedade e outro solicitando dados para as duas propriedades. Desta forma, o compilador saberá qual construtor chamar de acordo com o número de parâmetros passados aos atributo customizado.

Não existe uma restrição de apenas três atributos, crie quantos atributos julgar necessário, por exemplo, poderíamos ter um atributo que indicasse de que forma o campo chave primária seria preenchido, informando se por uma sequence + trigger, se por um campo auto incremento como no MySQL, entre outras informações (ver **BOX 5**).

Como a proposta é desenvolver um algoritmo para realizar a engenharia reserva não vamos aqui criar uma aplicação ou algo do tipo, ao invés disso vamos encapsular todas as regras de negócio em uma única classe e poderemos decidir de que

maneira utilizá-la em nossos projetos. Assim deixamos nosso algoritmo o mais abstrato possível, sem nenhum pré-requisito para ser executado, bastará adicionar as units criadas aqui nos projetos e usufruir das funcionalidades. Antes de iniciarmos a criação da nossa classe principal é importante falar sobre como ela funcionará.

BOX 4. Métodos sobrecarregados

Em orientação a objetos, uma sobrecarga refere-se aos métodos de uma classe que podem ser sobrecarregados em relação aos seus nomes, podendo vários métodos possuir o mesmo nome, porém os tipos de dados e a quantidade de parâmetros difere um método do outro. No Delphi para declarar um método sobrecarregado deve-se respeitar a regra já mencionada e adicionar ao final da declaração do método a palavra reservada overload.

BOX 5. Sequences e campos autoincremento

Geralmente campos que são chaves primárias em tabelas de banco de dados tem seus valores gerados automaticamente. Neste caso há duas abordagens mais comuns para realizar esta tarefa, uma delas é utilizar um objeto Sequence como os que existem no Oracle e Firebird e através de uma trigger de insert utilizar estes objetos para gerar os valores para os campos em questão. A outra abordagem é utilizar o conceito de campo auto incremento como no MySQL e SQLServer, onde ao marcar o campo como tal ele automaticamente gera os valores a medida que novos registros são inseridos na tabela.

DBExpress

Como a proposta é realizar a engenharia reversa de um banco de dados relacional qualquer, precisaremos utilizar algum componente de acesso a dados, afinal sem acesso ao banco não termos como mapeá-lo. A camada de conexão escolhida será o DBExpress por já ser um velho conhecido da comunidade Delphi e possuir driver para conexão aos principais SGBDs de mercado. Mas não se preocupe se sua aplicação não utilizar o DBExpress, lembre-se que o objetivo principal aqui é mapear o banco dados e gerar as classes de negócio para cada uma das tabelas contidas nele. Desta forma bastará criar uma nova aplicação em Delphi, adicionar um SQLConnection, realizar a conexão como seu banco de dados e utilizar nossa classe para realizar a engenharia reversa.

Essa extração dos metadados do banco de dados será feita utilizando dois métodos muito importantes presentes na classe TSQLConnection, são eles: GetTableNames e GetFieldNames que tem a função de respectivamente, extrair a lista de tabelas do banco de dados ao qual estamos conectados e extrair os campos de uma tabela deste mesmo banco de dados. No nosso caso, toda a infraestrutura estará baseada nestes métodos. Como temos que mapear para nossas classes o tipo de dados de um campo do banco de dados também utilizaremos internamente um componente TSQLDataSet que será instanciado dinamicamente. Isso se faz necessário pois precisamos identificar o tipo do campo para que possamos criar a propriedade em nossa classe do mesmo tipo ou de algum outro que suporte o dado contido neste campo do banco de dados.

Embora o objetivo seja criar uma classe que possa ser utilizada em qualquer aplicação Delphi seria importante também se esta classe pudesse interagir com componentes visuais, permitindo assim que possamos criar um interface para utilizar nossa framework. Por este motivo os métodos chaves para obter as tabelas, os campos e gerar as classes serão públicos, os dois primeiros retornarão as listas obtidas para que, se desejarmos, possamos adicionar estas listas em componentes listbox por exemplo.

Não somos obrigados a ficar limitados pelo uso dos componentes, se quisermos deixar nosso framework de engenharia reversa mais abstrato, poderíamos lançar mão das classes de conexão presentes da unit DBXCommon (ver **BOX 6**) e utilizar as classes TDBXConnection, TDBXCommand entre outras para realizar o mapeamento dos metadados dos nosso bancos de dados.

BOX 6. DBXCommon

O framework DBExpress passou por uma reformulação a partir do Delphi 2007 e foi evoluindo em versões posteriores. Uma série de novas classes, componentes, tipos e constantes foram adicionadas à VCL, entre elas a classes TDBXConnection responsável por criar uma conexão com banco de dados. Este objeto é criado através do padrão Abstract Factory utilizando a classe TDBXConnectionFactory. Além destas, há outras classes muito utilizadas como a TDBXCommand responsável por executar comandos SQL no bando de dados e a TDBXReader que permite obter o resultado de uma consulta.

Declarando a Classe Principal

Vamos então declarar a classe principal da nossa framework, a **Listagem 3** mostra em detalhes todos os métodos e Fields que utilizaremos para realizar a tarefa de gerar nossas classes.

Listagem 3. Declaração da Classe Principal

```
01. unit uMapDataBase;
02. interface
03. uses
04. SqlExpr, Classes, Dialogs;
05. type
06. TMapDataBase = class
07. strict private
08. class var FConnection: TSqlConnection;
09. class var FInstance: TMapDataBase;
10. constructor CreatePrivate;
11. private
12. FDataSet: TSQLDataSet;
13. FSaveDialog: TSaveDialog;
14. FTabelas: TStrings;
15. FCampos: TStrings;
16. FFields: TStrings;
17. FProperties: TStrings;
18. FOutputClass: TStrings;
19. function Captalizar(aTexto: String):String;
20. procedure InputCopyrigth(Tabela: String);
21. procedure GerarProperties(Tabela: String);
22. public
23. class function GetInstance(Connection: TSQLConnection): TMapDataBase;
24. constructor Create;
25. function ObterTabelas: TStrings;
26. function ObterCampos(Tabela: String): TStrings;
27. procedure GerarClasse(Tabela: String; Destino: String = '');
28. procedure GerarClasses;
29. end;
```

Note a presença de um método de classe na linha 23 além de um construtor privado na linha 10 e duas variáveis de classe nas linhas 8 e 9. Essas características estão presentes em objetos que implementam o padrão de projeto Singleton (ver **BOX 7**). Este padrão é a escolha certa em nossa classe pois quando fizermos uso desta biblioteca de funções em nossa aplicação, não justificará instanciar mais de um objeto do tipo TMapDataBase, basta uma única instância deste objeto para realizar a engenharia reversa de um banco de dados relacional. Outros exemplo de classe em que costumamos implementar o Singleton são: classes de usuários e conexão com banco de dados.

BOX 7. Singleton

O padrão de projeto Singleton é o mais simples e mais utilizado entre os padrões de projeto. Ele prega que uma classe que o implementa só poderá ter uma, e apenas uma, instância em toda aplicação e um ponto único de contato a esta.

O método GetInstance, que é na verdade quem faz o controle do Singleton, possui um parâmetro do tipo TSQLConnection. Fizemos isso para garantir que ao invocar o método para instanciar nossa classe e utilizar nossos recursos o usuário obrigatoriamente passe uma conexão com banco de dados. Se nossa objetivo é realizar engenharia reversa de um banco de dados o mínimo que precisamos ter é uma conexão a este banco de dados. Este método pode e deve ser melhorado, ao receber como parâmetro uma instância de um TSQLConnection nós poderíamos de imediato testar a conexão com o banco de dados e se está não for bem sucedida interromper neste momento a criação do nosso objeto e informar ao usuário sobre o problema.

Além dos métodos para implementar o padrão Singleton nós temos quatro métodos públicos para realizar de fato o trabalho de geração das classes, na linha 25 o método "ObterTableas" que retorna a lista de tabelas para quem o invocar em um TStrings para que, como já citado anteriormente, possamos se desejarmos adicionar esta lista de tabelas em um componente visual como um Listbox. Na linha 26 temos o método "ObterCampos", que tem função análoga ao método "ObterTabelas", porém desta vez retorna a lista de campos de uma determinada tabela passada como parâmetro. Por fim na lista 27 e 28 temos os métodos que geram a unit final contendo a classe que representa a estrutura de uma determinada tabela. O Método "GerarClasse" recebe o nome de uma tabela e gerar a unit com a classe para esta tabela, opcionalmente você poderá informar um caminho para onde a classe deverá ser gerada, se nada for informado uma caixa de diálogo será exibida para o usuário selecionar o destino. Já o método "GerarClasses" nada mais é do que um loop na lista de tabelas e para cada uma delas invoca o método "GerarClasse", desta forma com um único clique de botão podemos gerar todas as classes de uma vez só. Muito útil isso.

Bem mas não são só estes métodos que fazem a mágica acontecer, estes são os métodos públicos, que serão acessíveis por quem fizer uso da nosso framework. Aplicamos aqui o conceito do encapsulamento e colocamos no escopo privado tudo aquilo que não é necessário expor a quem utilizará a classe. Assim os detalhes de funcionamento interno deste mecanismo de engenharia reversa ficam privados a classe evitando assim chamadas desnecessárias ou equivocadas aos métodos. Os fields e métodos privados estão descritos entre as linhas 12 a 21. É possível notar a presença de muitos fields do tipo TStrings, são eles que armazenarão os comandos que serão gerados para no fim dar vida a nossa unit. Não podemos esquecer que uma unit na verdade é um arquivo de texto com extensão ".pas" que possui uma sintaxe própria do Delphi, então o que de fato faremos é criar este arquivo respeitando essa sintaxe e utilizando os dados extraídos do banco de dados para compor nossa classe e propriedades.

Temos então uma lista para cada parte da nossa unit, em FTabelas manteremos a lista de tabelas que será retornada pelo método "ObterTabelas", além de retornarmos para quem invocar o método também guardaremos a lista em FTabelas. Em FCampos serão guardados os campos também retornados pelo métodos "ObterCampos", note que neste caso são os campos do banco de dados não os Fields da classe que será gerada. Estes serão mantidos na lista FFields já no padrão de nomenclatura das classes Delphi, ou seja, com a letra F na frente do nome. Isso será visto na implementação.

A lista FProperties manterá a lista de propriedades que estarão presente em nossa classe, também no padrão de declaração do Delphi com a palavra **property** antes do nome da propriedade e a declaração dos mecanismos de acesso os famosos Get e Set. Por último mais não menos importante, temos o item FOutputClass, ele que fará a união de todos os "pedaços" da nossa unit para gerar um arquivo final respeitando a sintaxe do Delphi e fará a geração do arquivo em disco.

Além destas listas temos outros métodos e fields acessórios para apoiar na geração das nossas classes. O item FDataSet é utilizado para listar os campos do banco de dados, mas não confunda com o método "ObterCampos". Este último lista apenas os nomes dos campos e não seus tipos e o objetivo dele é justamente retornar a lista de campos para, por exemplo, popular um componente listbox caso desejarmos montar uma interface para interagir com nossa classe. Para gerar as propriedades das classes que criaremos com nosso framework, precisamos saber o tipo de cada campo da tabela que estamos mapeando e é ai que entra o FDataSet que é na verdade um componente TSQLDataSet. Montaremos um comando de select para retornar todos os campos, porém, sem nenhum registro já que o objetivo é apenas mapear os campos e tipos. O Field FSaveDialog dispensa comentários, ele será acionado sempre que solicitar a geração de uma classe e não informarmos um destino para salvar o arquivo.

Além destes Fields que acabamos de comentar temos também alguns método como o "InputCopyright" e "Capitalizar". O método "InputCopyright" insere um texto com detalhes de como a classe foi gerada e as informações do autor, tudo isso em formato de comentário do Delphi. Isso é uma abordagem muito comum quando se programa orientado a objetos ou quando criarmos componentes. Já o método "Capitalizar" nada mais faz do que colocar a primeira letra do texto passado em maiúscula e o restante em minúscula para seguir um padrão de nomenclatura. Isso se faz necessário pois geralmente as tabelas e campos em um banco de dados são criadas em caixa alta e se gerássemos nossas classes e propriedades destas forma o código ficaria ruim para ler. Este método "Capitalizar" poderia ser melhorado para algo mais abrangente como retirar hifens, underlines e qualquer outro caractere ou prefixo que por ventura as tabelas e campos possuam.

Para fechar a declaração da nossa classe temos o último método, "GerarProperties". Este método tem por finalidade obter os campos da tabela que está sendo mapeada e preencher o Field FProperties e FFields. É neste método que utilizamos o componentes TSQLDataSet através do fields FDataSet para listar o tipo de cada um dos campos.

Implementação da Classe Principal

O objetivo de cada um dos métodos e Fields declarados em nossa classe foi devidamente explicado, alguns métodos dispensam comentários como é o caso do construtor e método "GetIsntance" pois seus códigos falam por si só. O mesmo vale para os métodos "Capitalizar" e "InputCopyright". Notem que como dito no início do artigo utilizamos um código muito simples para resolver um problema complexo ou ao menos evita uma tarefa trabalhosa. Os métodos baseiam-se em loops e concatenação de texto para criar em memória dentro do Field FOutputClass nossa classe de negócio, respeitando a sintaxe da linguagem Delphi.

Acho importante apenas ressaltar o que foi feito nos métodos principais, começando com o mais importante: "GerarClasses" que pode ser visto na **Listagem 4**.

Nota

Nosso Framework não gerará os métodos Get e Set para acessar os Fields das propriedades, faramos acesso direto aos Fields, com isso simplificamos o algoritmo de criação. Você poderá tratar pontualmente e alterar as classes depois de geradas para implementar os métodos Get e Set onde estes se fizerem necessários.

Listagem 4. Método GerarClasse

```
procedure TMapDataBase.GerarClasse(Tabela: String; Destino: String = '');
var
  Diretorio,cTabela: String;
begin
  FOutputClass.Clear;
  if Destino <> EmptyStr then
    Diretorio := Destino
  else
    if FSaveDialog.Execute then
      Diretorio := ExtractFilePath(FSaveDialog.FileName)
    else
      Exit;

  if FTabelas.Count = 0 then FTabelas := ObterTabelas;
  FCampos := ObterCampos(Tabela);
  cTabela := Captalizar(Tabela);
  InputCopyrigth(cTabela);
  FOutputClass.Add('unit u'+cTabela+';');
  FOutputClass.Add('');
  FOutputClass.Add('interface');
  FOutputClass.Add('');
  FOutputClass.Add('uses');
  FOutputClass.Add('  uAtributos;');
  FOutputClass.Add('');
  FOutputClass.Add('type');
  FOutputClass.Add('  [TTable('+ QuotedStr(cTabela)+')]');
  FOutputClass.Add('  T'+cTabela+' = class');
  FOutputClass.Add('  private');
  FOutputClass.Add(FFields.Text);
  FOutputClass.Add('  public ');
  FOutputClass.Add(FProperties.Text);
  FOutputClass.Add('  end;');
  FOutputClass.Add('');
  FOutputClass.Add('implementation');
  FOutputClass.Add('');
  FOutputClass.Add('');
  FOutputClass.Add('');
  FOutputClass.Add('end.');
  FOutputClass.SaveToFile(Diretorio+'\u'+cTabela+'.pas');
end;
```

Este método recebe como parâmetro o nome da tabela que vamos mapear e opcionalmente o destino onde o arquivo ".pas" será gerado. Nós iniciamos o método limpando o conteúdo de FOutputClass, pois lembre-se que podemos invocar este método em loop e não queremos correr o risco de ir acumulando os códigos das units a medida que são geradas. Após este passo testamos a questão do diretório de salvamento e caso nenhum caminho tenha sido informado executamos o componente SaveDialog para que o usuário informe o diretório onde quer salvar a unit que será gerada. Em seguida testamos se as tabelas já foram obtidas do banco de dados e caso ainda não tenham sido, invocamos o método "ObterTabelas". Lembre-se que o usuário pode utilizar a nossa classe e não necessariamente obter as tabelas para exibir em um componente, ele pode simplesmente invocar o método "GerarClasse" diretamente, por isso precisamos testar se as tabelas já foram extraídas do banco de dados.

Feito isso acionamos o método "ObterCampos" e aqui tem um detalhe importante, veja na **Listagem 5** que na implementação do método "ObterCampos" é que chamamos o método "GerarProperties".

Listagem 5. Método ObterCampos

```
function TMapDataBase.ObterCampos(Tabela: String): TStrings;
begin
  FCampos.Clear;
  if FTabelas.IndexOf(Tabela) < 0 then
    raise Exception.Create('Esta tabela não pertence ao banco que está sendo
    mapeado');
  FConnection.GetFieldNames(Tabela,FCampos);
  GerarProperties(Tabela);
  Result := FCampos;
end;
```

Fazemos isso para garantir que o conteúdo dos fields FCampos e FProperties sejam iguais e evitar que geremos uma classe com os campos de uma tabela e as propriedade de outra. Outro cuidado que tomamos no método "ObterCampos" é testar o nome da tabela passada como parâmetro para evitar que seja solicitado obter campos de uma tabela que não faz parte do banco de dados que está sendo mapeado. Já que este método é público e pode ser invocado pelo usuário a qualquer momento, temos que garantir que somente sejam aceitas tabelas que constem em nossa lista de tabelas.

Voltando ao método "GerarClasse", após obter as tabelas e campos iniciamos de fato a construção da nossa classe adicionando ao Field FOutputClass as linhas de códigos necessárias para criação de uma classe no Delphi. Declaramos a palavra reservada unit mais o nome da tabela que está sendo mapeada precedida da letra "u" de unit. Na sequência adicionamos a seção interface, incluímos na seção uses a unit uAtributos já que nossa classe irá possuir os atributos customizados que criamos anteriormente. Seguimos incluindo a palavra reservada "Type" e então de fato declaramos a nossa classe. Como já citado é comum iniciarmos o nome das classes no Delphi com a letra T por isso o nome da nossa classe é gerado concatenando a letra T mais o nome da tabela que está sendo mapeada seguida da declaração da palavra reservada "class". Note que na linha imediatamente anterior declaramos o atributo TTable e passamos o nome da tabela original no banco de dados para ele, exatamente como descrito no início deste artigo.

Continuando na construção da nossa classe declaramos o escopo private e logo abaixo incluímos o conteúdo do Field FFields que já foi preenchido pelo método "GerarProperties". Neste ponto serão adicionado todos os Fields já mapeados e formatados no padrão Campo:Tipo. Seguimos adicionando o escopo public e da mesma forma que fizemos com o Field incluímos aqui o conteúdo do Field FProperties que contém todas as propriedades, também já na sintaxe Delphi que foram preparadas no método "GerarProperties" que está descrito na **Listagem 6.**

Listagem 6. Método Gerar Properties

```
procedure TMapDataBase.GerarProperties(Tabela: String);
var
  I: Integer;
  Prop, Campo, Tipo: String;
begin
  FProperties.Clear;
  FFields.Clear;
  FDataSet.SQLConnection := FConnection;
  FDataSet.CommandText := 'Select * from '+Tabela+' where 1=0';
  FDataSet.Active := True;
  for I := 0 to FDataSet.FieldList.Count - 1 do
  begin
    Campo := Captalizar(FDataSet.Fields[I].FieldName);
    Prop :='property '+Campo;
    if FDataSet.Fields[I].DataType in [ftString, ftMemo] then
      Tipo := ':String'
    else
    if FDataSet.Fields[I].DataType = ftBoolean then
      Tipo := ':Boolean'
    else
    if FDataSet.Fields[I].DataType in [ftSmallint, ftInteger, ftWord] then
      Tipo := ':Integer'
    else
    if FDataSet.Fields[I].DataType in [ftFloat, ftCurrency, ftBCD] then
      Tipo := ':Currency'
    else
    if FDataSet.Fields[I].DataType in [ftDate, ftTime, ftDateTime, ftTimeStamp] then
      Tipo := ':String'
    else
      Tipo := ':Double';

    Prop := Prop+Tipo+' read F'+Campo+' write F'+Campo+';';
    FProperties.Add('    [TColumn('+QuotedStr(Campo)+')]');
    FProperties.Add('    '+Prop);
    FFields.Add('    F'+Campo+Tipo+';');
  end;
  FDataSet.Close;
end;
```

Por falar em método "GerarProperties" notem que é neste método que determinamos os tipos das propriedades com base nos tipos dos campos do banco de dados. Utilizamos para isso o componente TSQLDataSet com um select na tabela que estamos mapeando porém sem retornar registros. Não estamos interessados nos dados mas sim na estrutura da tabela, logo retornar

a tabela vazia é suficiente para atender nossa necessidade que neste caso é criar os TFields dentro do componente para que possamos acessá-los. Veja que após ativar o FDataSet iniciamos um loop na lista de TFields criada é para cada TField encontrado nós capturamos seu nome para montar nossa propriedade e campo da classe. Em seguida testamos a propriedade DataType do TField para identificar seu tipo. É neste ponto que nós realizamos um De -> Para dos tipos do banco de dados para os tipos das propriedades da nossa classe. Esse código pode ser ajustado para as mais diversas realidades, por exemplo, aqui eu mapeei os campos de data e hora para string, porém pode-se dar um tratamento diferenciado se quiser.

Aqui também lançamos mão do recurso de Custom Attributes, declarando o atributo customizado TColumn imediatamente antes de adicionar a string contendo a declaração da propriedade que estamos gerando. Esta abordagem é muito semelhante a que fizemos com o atributo TTable e como já citado, você tem total liberdade para criar novos atributos e adaptar este código para suprir necessidades específicas do seu dia a dia.

Voltando ao método "GerarClasse", após inserir as propriedades fechamos a declaração da classe com o "end;", incluímos a declaração implemetation e encerramos nossa classe com o "end". A última linha do método é justamente o comando que salva no disco dentro do diretório informado pelo usuário a unit que acabamos de montar que contem a nossa classe mapeada com base na tabela do banco de dados. Note que salvamos o arquivo utilizando no nome o mesmo conceito empregado para nomear a unit, adicionando a letra "u" antes do nome da tabela.

Nossa classe está finalizada e já podemos utilizá-la para gerar classes de negócio com base em banco de dados relacionais. O objetivo do artigo não era criar um mecanismo de persistência, mas sim desenvolver uma solução para o trabalho que é criar classes de negócio para banco de dados muito grandes o que tomaria muito nosso tempo. Agora que já lhe apresentei o conceito e que seu tempo será poupado, invista este tempo extra em estender este projeto. Adicione novos atributos, refine o mapeamento dos campos, crie uma interface para acionar os métodos da nossa classe. Se desejar melhore o método "GerarProperties" para ao invés de acessar os fields diretamente sejam gerados também os métodos Get e Set. Enfim, muito ainda pode ser feito para deixar nossa framework de engenharia reversa mais eficiente, então mãos à obra.

Autor

Rodrigo Mourão
rodrigomourao@rodrigomourao.com.br
http://www.rodrigomourao.com.br

Gestor de TI com especialização em gerência de projetos (PMI) e Governança de TI (ITIL e COBIT) pelo instituto Infnet. Graduando-se em Gestão de TI focada em negócios. Grande entusiasta do desenvolvimento orientado a objetos, padrões de projetos, modelagem de processos de negócio e metodologias ágeis. Atualmente encontra-se como Gerente de TI e processos no Grupo Editorial Nacional, líder no segmento de editorial CTP (científicos, técnicos e profissionais) no Brasil.

Links:

Blog Rodrigo Mourão
http://www.rodrigomourao.com.br
http://fb.com/blogrodrigomourao

Você gostou deste artigo?

Dê seu voto em ***www.devmedia.com.br/clubedelphi/feedback***

Ajude-nos a manter a qualidade da revista!

Refatoração no Delphi XE5

Mantendo seu código sempre limpo e organizado usando refatoração contínua

Este artigo é do tipo mentoring

saiba mais: www.devmedia.com.br/mentoring-saibamais

Cenário

Conforme sistemas de software são evoluídos, modificados e adaptados de acordo a adição de novas funcionalidades, seu código e outros artefatos envolvidos se tornam mais complexos e se afastam de seu objetivo original e perdem qualidade. Metodologias e ferramentas não resolvem este problema, pois sua utilização normalmente auxilia na aceleração no desenvolvimento. Em muitos projetos, boa parte do custo é dedicada à sua manutenção. Dessa forma, é necessário facilitar a manutenibilidade e legibilidade do código, melhorando sua qualidade interna. Uma das técnicas usadas para solucionar esse problema é conhecida como Refatoração. Este artigo apresenta os fundamentos envolvidos na prática da refatoração, bem como mostra um estudo de caso através de uma aplicação prática feita no Delphi XE5, mostrando ainda o suporte que o IDE oferece através do menu Refactor.

Refatoração é o processo de modificar um sistema de software sem alterar o comportamento externo do código, melhorando sua estrutura interna. É uma forma disciplinada de limpar minimizando as chances de introduzir erros. Quando se refatora está se melhorando o design do código depois de ele ter sido feito. Primeiro um bom projeto deveria ser feito, a seguir a sua codificação, mas nem sempre é assim que as coisas acontecem. Com a evolução do sistema de software, mudanças ocorrerão, e a sua estrutura e integridade original diminuem. Dessa forma, essas transformações, no caso as refatorações, podem melhorar a estrutura do projeto. Embora a reestruturação crie novas versões que implementam ou propõem mudanças no objetivo final do sistema, normalmente não envolvem modificações referentes a novos requisitos, ou seja, não visam implementar novas funcionalidades no sistema de software. No entanto, podem conduzir a observar melhor o objetivo do sistema, já que sugerem mudanças que poderem melhorar sua estrutura interna. Nesse sentido, é necessário identificar locais no código ou outros artefatos que precisam ter sua estrutura modificada, seja para melhorar sua legibilidade, melhorar o projeto, tornar mais simples de evoluir e de manter. Tais estruturas passíveis de refatoração são também conhecidas como "bad smells", ou problemas (shortcomings). Um bad smell sugere que existem deficiências que podem ser sanadas ou melhoradas, porém é necessário que a aplicação continue apresentando o mesmo comportamento externo após a transformação interna.

Outro ponto a ser considerado no estudo da refatoração é o uso de Padrões de Projeto, ou Design Patterns, amplamente utilizados no projeto de arquiteturas de software, pois representam soluções gerais, reutilizáveis, testadas e documentadas, resolvendo problemas recorrentes de programação orientada a objetos. Muitas vezes, no entanto, padrões de projeto tendem a ser utilizados de forma antecipada em um projeto. Já com refatoração, você parte de um mau design (muitas vezes do caos absoluto), refatora e obtém um código bem projetado. Cada refatoração consiste de um passo simples, pequeno. Você move um campo para outra classe, move um método para uma classe superior na hierarquia, extrai uma classe nova a partir de outra existente etc. Cada pequeno passo vai melhorando de forma incrementa o design do código como um todo.

Existem várias motivações para o uso de refatorações. Elas tornam mais fácil a adição de novo código. Quando se inclui uma nova funcionalidade em um sistema de software, pode-se programar sem se preocupar o quão bem se encaixa no projeto existente, ou pode modificar o projeto existente a fim de melhor implementar a funcionalidade.

No primeiro caso, ocorre um débito de projeto, o qual pode ser resolvido mais tarde com refatoração. Além disso, as refatorações melhoraram um projeto de código existente, no momento que tornam o código mais simples, claro e mais fácil de trabalhar, manter e evoluir. A refatoração contínua envolve procurar constantemente por problemas e removê-los imediatamente. E finalmente, refatorações ajudam a se obter um melhor entendimento do código. Se um código está difícil de ser compreendido, programadores normalmente comentam este código para tornar mais clara sua intenção. Se o código não está claro, o problema precisa ser removido através de refatoração.

Refatoração é uma atividade vital em abordagens ágeis de desenvolvimento. Uma transformação na estrutura pode ser frequentemente relacionada à sua aparência (por exemplo, melhorar a legibilidade), mas pode ser uma alteração no design da aplicação. Refatorações podem melhorar a qualidade de um software, por exemplo, extensibilidade, modularidade, maior reutilização de código, menor complexidade, manutenção mais fácil, eficiência etc. Podemos dividir as refatorações em duas grandes categorias, as primitivas e as formados por composição. Há ainda refatorações mais complexas, as compostas que tem como resultado da transformação a aplicação de um Padrão de Projeto (Design Pattern). Uma refatoração primitiva seria, por exemplo, renomear um objeto, mover um método etc. As refatorações mais complexas são normalmente obtidas pela composição de várias primitivas. O processo de refatoração consiste em uma série de atividades distintas, incluindo: identificar onde um software precisa ser refatorado, determinar qual refatoração pode ser aplicada, garantir que o comportamento se mantém, aplicar a refatoração, medir o impacto do refactoring nas características do software, como complexidade, usabilidade, custo, esforço etc. Manter a consistência entre o código refatorado e outros artefatos do sistema, como diagramas UML, documentação em HTML, PDF, especificação de requisitos, testes etc. Identificar bad smells no código para criar oportunidades de refatoração é uma técnica bastante interessante e difundida atualmente. De acordo com Kent Beck, bad smells ("maus cheiros") são estruturas no código que sugerem a possibilidade de refatoração. Alguns exemplos de problemas em estruturas de código são: código duplicado, métodos longos, classes longas, métodos com muitos parâmetros, entre outros. Podemos usar métricas de software para, por exemplo, identificar falhas no design de um software que apresenta classes de um framework com um alto grau de acoplamento, reduzindo assim o reaproveitamento. Neste caso, poderia se usar refatoração para padrões de projeto, a fim de melhorar o design. Testes ou gráficos de dependência podem ajudar a medir a eficiência dos refactorings. Outro ponto importante a ser considerado são as pré e pós condições das atividades de refatoração. Por exemplo, uma pré-condição de um sistema de tempo real, como uma aplicação Web de E-Commerce, é o tempo de resposta ao usuário. De nada adianta melhorar a legibilidade do código e design se ao final tivermos uma queda no tempo de resposta, pois nesse caso, performance é uma pré-condição. O mesmo vale para sistemas embarcados, por exemplo, aplicações que rodam em smartphones, elas precisam consumir pouca memória e bateria, outra pré-condição.

Antes de entrarmos em um estudo de caso, vamos fazer uma análise da refatoração dentro do contexto de desenvolvimento Delphi, rebuscando a história das linguagens RAD. Antes do surgimento do Delphi 1, que foi um marco para o desenvolvimento visual na época, tínhamos apenas o Visual Basic como forte concorrente. Foi exatamente nesta época que surgiu o conceito de "desenvolvimento visual", ou as "linguagens visuais", incluindo aí o então IDE do Visual Studio a seguir. O termo "visual" se originou do fato de que as linguagens de programação não tinham um ambiente gráfico de design, basicamente você programava num editor de textos "às escuras", compilava, rodava e torcia para que os elementos gráficos ficassem organizados como previsto no código. Assim era feito em ambientes como o Turbo Pascal e o Turbo C, e começou a mudar com a introdução do Turbo Vision, o tataravô do Delphi. Esse "apelo" trazido pelas linguagens visuais, como Delphi e VB, tornaram o desenvolvimento extremamente rápido, muito rápido. Aquela barra de menu que se levava duas semanas para construir estava ali pronta na caixa de ferramentas, basta fazer o "drag & drop" e pronto. Antes mesmo de compilar, você já tem o visual da interface de usuário. Esse desenvolvimento rápido introduziu outro termo amplamente utilizado até hoje, o RAD – Rapid Application Development, o desenvolvimento rápido de aplicações. Aqui entra o fator crítico e o elo dos "studios" (RAD, Visual etc.) com a Refatoração: a programação seguindo esta abordagem RAD permite que muitos desenvolvedores abram mão de boas práticas de orientação a objetos e padrões de projeto para obter um mesmo resultado mais rapidamente, seja por falta de conhecimento, pressão devido a prazos curtos ou mesmo para reduzir custos abrindo mão da qualidade. O sistema parece funcionar bem por fora, mas por dentro está mal projetado, ou não tem projeto, ou está um verdadeiro caos que nem mesmo o programador original o entende. Todo o código está em um único formulário ou data module. Não existem classes de negócio, e as poucas que existem estão fortemente acopladas. Conforme a aplicação cresce, mais componentes, código e funcionalidades (e bugs, muitos deles) são adicionados ao projeto original. Para muitas empresas, se não a maioria delas (exceto startups), o foco principal não é criar software rapidamente, mas software que seja fácil de ser mantido ao longo do tempo, pois isso reduz custos a longo prazo. A refatoração, neste caso, entra como a ponte, a chave, o elo, a solução para se criar software com código sempre limpo e funcional, preparado para mudanças, que sempre existirão.

Estudo de Caso

Após o domínio dos fundamentos e técnicas relacionadas à prática de refatoração, vamos criar uma aplicação prática no Delphi XE 5. Através do menu Refactor (**Figura 1**), podemos observar o suporte do próprio IDE para as principais refatorações catalogadas.

Vamos considerar um simples exemplo para entender como funciona o processo de refatoração. É um programa que calcula e imprimi relatórios de diárias de locação de filmes de um cliente em uma locadora (**Figura 2**).

Listagem 1. A classe Filme representa os dados de um filme

```
unit GPauli.Sistemas.Locadora.Framework.Filme;
interface
type
 Filme = class
  public const Catalogo: integer = 0;
  public const Lancamento: integer = 1;
  public const Infantil: integer = 2;
  private _titulo: string;
  private _codigoPreco: integer;
  public function getCodigoPreco(): integer;
  public function getTitulo(): string;
  public procedure setCodigoPreco(codigoPreco: integer);
  public constructor Create(titulo: string; codigoPreco: integer);
 end;
implementation
{ Filme }
//implementação disponível no código fonte para download
end.
```

Listagem 2. A classe Locacao representa um cliente alugando um filme

```
unit GPauli.Sistemas.Locadora.Framework.Locacao;
interface
uses
 GPauli.Sistemas.Locadora.Framework.Filme;
type
 Locacao = class
  private _filme: Filme;
  private _diasLocados: integer;
  public function getFilme(): Filme;
  public function getDiasLocados(): integer;
  public constructor Create(filme: Filme; diasLocados: integer);
 end;
implementation
{ Locacao }
//implementação disponível no código fonte para download
end.
```

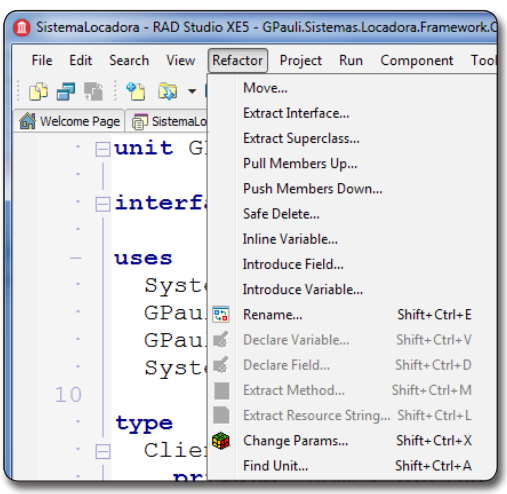


Figura 1. Menu Refactor

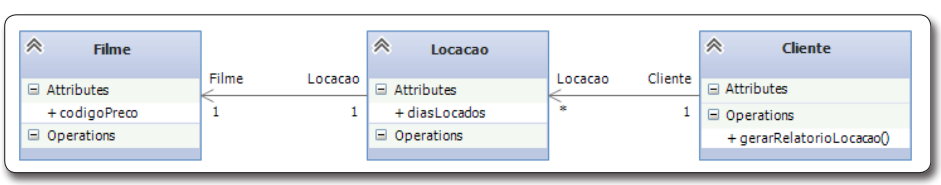


Figura 2. Framework de sistema de locação

Ele indica quais os filmes foram alugados por determinado cliente e por quanto tempo. Existem três tipos de filmes: catálogo, infantil e lançamento. O programa pode considerar a pontuação do cliente, de acordo com locação de lançamentos. Os códigos das classes são apresentados nas **Listagens 1** a **3**. Para testar estas classes, podemos criar uma interface de usuário FMX ou VCL, ou ainda criar testes unitários no próprio IDE. Para este exemplo de estudo de caso, pode-se utilizar testes unitários ou uma aplicação de Console simples (**Listagem 4**).

O resultado da execução pode ser visto na **Listagem 5**. Para aplicar as refatorações apresentadas nestes artigo, quando indicadas, basta selecionar o código desejado (método, bloco de código, objeto) e então escolher a opção respectiva no menu, ou ainda, no menu de contexto (clique de direita).

Listagem 3. A classe Cliente representa dados do locatário e gera um relatório de locação

```
unit GPauli.Sistemas.Locadora.Framework.Cliente;
interface
uses
 System.Generics.Collections,
 GPauli.Sistemas.Locadora.Framework.Locacao,
 GPauli.Sistemas.Locadora.Framework.Filme,
 System.SysUtils;
type
 Cliente = class
  private _nome: string;
  private _locacoes: TList<Locacao>;
  public function getNome(): string;
  public constructor Create(nome: string);
  public destructor Destroy(); override;
  public procedure addLocacao(locacao: Locacao);
  public function gerarRelatorioLocacao(): string;
 end;
implementation
{ Cliente }
//implementação disponível no código fonte para download
end;

end.
```

Este programa aparentemente não apresenta nenhum problema, por ter uma funcionalidade simples. Porém, não está com um bom design nem orientado a objetos. Em sistemas mais complexos, o método que imprimi a declaração teria muitas responsabilidades, está muito longo. Muitas de suas atribuições poderiam ou deveriam ser feitas por outras classes. O design deste pequeno programa está pobre, difícil de ser compreendido e alterado. Por exemplo, imagine que o cliente agora deseja que a declaração seja exibida em uma página Web, logo é necessário modificar a saída para gerar o relatório para outro formato. Olhando para a complexidade do código, dizemos que é impossível reutilizar qualquer porção do código. A única saída seria reescrever um novo método e acabar repetindo muito do que já existe. Você poderia copiar o código e aplicar as alterações necessárias para obter uma

saída em formato compreensível na Web. Porém, o que ocorre se uma mudança nas regras for necessária? É necessário fixar ambos os métodos que geram relatórios de locação. Essas regras estão implementadas todas em um único método, e são lá que elas devem ser modificadas. Essas consistências devem também ser garantidas no método duplicado. Além disso, como as regras crescem em complexidade, será mais difícil descobrir onde aplicar uma mudança e como fazer isso sem cometer um erro.

Listagem 4. Aplicação de exemplo para testar sistema de locação

```
program SistemaLocadora;
{$APPTYPE CONSOLE}
{$R *.res}
uses
  System.SysUtils,
  GPauli.Sistemas.Locadora.Framework.Filme in
'GPauli.Sistemas.Locadora.Framework.Filme.pas',
  GPauli.Sistemas.Locadora.Framework.Locacao in
'GPauli.Sistemas.Locadora.Framework.Locacao.pas',
  GPauli.Sistemas.Locadora.Framework.Cliente in
'GPauli.Sistemas.Locadora.Framework.Cliente.pas';
var
  flm1,flm2: Filme;
  loc1,loc2: Locacao;
  cli: Cliente;
begin
  cli := Cliente.Create('Guinther Pauli');
  flm1 := Filme.Create('Whale Rider',flm1.Catalogo);
  flm2 := Filme.Create('Seabiscuit',flm2.Lancamento);
  loc1 := Locacao.Create(flm1,3);
  loc2 := Locacao.Create(flm2,4);
  cli.addLocacao(loc1);
  cli.addLocacao(loc2);
  Writeln(cli.gerarRelatorioLocacao());
  Readln;
end.
```

Listagem 5. Relatório de locação

```
Relatório de locação para cliente Guinther Pauli
Filme: Whale Rider - 3,5 pontos
Filme: Seabiscuit - 12 pontos
Total acumulado é 15,5 pontos
Você ganhou 3 pontos pela locação
```

Lembre-se do velho ditado da engenharia: "Se não está quebrado, não conserte". A funcionalidade do programa não pode ser quebrada, porém está sendo muito difícil implementar as mudanças que os usuários querem. É nesse ponto que entra a refatoração. Quando for necessário incluir uma nova funcionalidade em um código existente, porém a estrutura do programa não está apropriada para receber a nova característica, primeiro refatore o programa para torná-lo mais simples para ser modificado, então adicione a funcionalidade. O primeiro passo rumo à refatoração é criar testes, eles vão garantir que bugs não sejam introduzidos durante a refatoração. Um possível teste para o dado cenário poderia incluir a criação de alguns clientes, simular algumas locações e verificar as declarações obtidas. Estes testes devem ser rodados sempre que uma refatoração for feita. O próximo passo na refatoração é decompor em pequenas partes o método que gera declarações. O objetivo é mover partes do código para novos métodos e classes, menores. A refatoração Extract Method é uma boa candidata a ser aplicada neste caso. Um cuidado deve ser tomado para que bugs não sejam introduzidos, por exemplo, com a utilização de variáveis que dependem do escopo de onde estão sendo executadas. Por isso é importante lembrar também que o processo de refatoração deve ser realizado em pequenos passos (baby steps, ou small steps).

Veja que a refatoração não deve ser usada em princípio para corrigir bugs, mas como uma forma de ajudar a identificá-los. Se um sistema de software está repleto de bugs e está difícil de estabilizá-lo, talvez uma melhor opção seria reescrevê-lo do zero, o que não é uma decisão fácil. A refatoração neste caso também serve como uma alternativa ao chamado "grande projeto antecipado". Nesse cenário não se cria inicialmente um grande projeto, que tente prever todas as possibilidades de mudanças que possam ocorrer no sistema de software. Ao invés disso, criam-se pequenas porções do software bem desenhadas, como sugere a XP (Extreme Programming) (ver **BOX 1**). A refatoração nesse caso dá ênfase nas mudanças. Utiliza-se a abordagem incremental, porém não se tenta achar a solução para tudo. Ao desenvolver de forma evolutiva, o conceito de solução pode mudar, ou melhorar, e com a refatoração isso não se torna um problema.

BOX 1. Refatoração e Metodologias Ágeis

Metodologias tradicionais de engenharia de software, focadas em planejamento, têm sido utilizadas ao longo do tempo. O grande problema é que elas são muito burocráticas, dão ênfase em contratos fixos, documentação excessiva, vasto planejamento antecipado (prematuro). Essa abordagem não condiz mais com a realidade da maioria dos projetos na área de tecnologia, onde novos requisitos são criados, modificados ou mesmo removidos após um planejamento inicial. Processos e metodologias modernas, como as metodologias ágeis de desenvolvimento, são evolutivas por natureza. Organizam o trabalho de forma interativa, realizado em pequenas partes, com pequenas entregas (design, código, teste, documentação), liberadas aos poucos, ao invés de um único release enorme. Dão maior ênfase na colaboração e comunicação aberta e fluente entre stakeholders, clientes e membros do time, minimizando as chances de insucesso em projetos. Os princípios básicos da abordagem ágil sugerem que indivíduos e interações são mais importantes que processos e ferramentas, trabalho deve ser feito mais focado no software em si que em uma documentação abrangente, colaboração do cliente mais que negociação de contrato e, principalmente, responder a mudanças mais que seguir um plano. O problema é que as abordagens tradicionais de engenharia de software se baseiam em outros ramos de engenharia, como a civil e mecânica, onde os custos com planejamento são menores. Em software, estima-se que até 50% de custos são gastos com design e planejamento. Projetos com um escopo invariável são raros, e se requisitos constantemente mudam, há um grande desperdício de esforço com análise que é descartada em virtude disso. Trabalho, tempo e dinheiro são preciosos são perdidos. Uma abordagem ágil garante que a equipe e projeto sejam capazes de se adaptar às mudanças que ocorrerão, dentro de um contexto evolutivo.

Uma metodologias ágil, como a XP – Extremme Programing, prega cinco valores principais: comunicação, feedback, simplicidade, coragem e respeito, com foco maior em disciplina do que processos. Já Scrum prega um desenvolvimento ágil baseado em iterações chamadas sprints e reuniões de acompanhamento frequentes (meetings). Aplicações reais precisam continuamente

ser evoluídas, novos requisitos implementados, melhorias devem ser feitas no projeto existente (design), algumas vezes é necessário mudar de tecnologia ou plataforma, corrigir bugs, otimizar e melhorar a performance. Isso remete a uma pergunta: O que pode ser considerado um "bom software"? Um bom software é aquele que funciona como planejado, mas depende do seu negócio. Para pequenas companhias (startups), bons softwares devem ser criados rapidamente. Para grandes corporações, o foco na construção de um software é torná-lo fácil de ser mantido ao longo do tempo. Nesse sentido, é necessário tomar cuidado com bons princípios de design, como criar componentes com baixo acoplamento, reutilizáveis, que garantam que alterações em componentes não afetam outros, que objetos tenham responsabilidades únicas, separadas, partes facilmente substituíveis, complexidade encapsulada. Esses são princípios básicos da programação orientada a objetos que, vistos dentro de um contexto de uma abordagem ágil evolutiva, vão tornar o sistema de software mais fácil de testar, evoluir, manter e entender. Padrões de Projeto (Design Patterns) ajudam a criar bons designs de software, já o uso de Refatoração contínua garante código claro, limpo e funcional em equipes que programam de forma mais ágil, sem se preocupar em um primeiro momento com muitos aspectos de design, usando, por exemplo, Padrões de Projeto. Padrões de Projeto são soluções reutilizáveis para problemas recorrentes no desenvolvimento orientado a objetos. Aplicar padrões antecipadamente em um projeto requer mais tempo e aumenta custos, ao mesmo tempo em que aumenta sua complexidade. Requisitos são modificados, criados ou mesmo eliminados, conforme projetos evoluem. Tentar aplicar padrões antecipadamente para prever essas mudanças é um problema conhecido como excesso de engenharia. Já projeto em escassez reduz custos, pois o sistema de software é criado rapidamente devido à falta de tempo, conhecimento ou necessidade de se adicionar funcionalidades em curto prazo. Ao longo do tempo, armazena-se um débito de projeto, que torna o sistema de software muito difícil de ser mantido e evoluído. É nesse contexto que entram as refatorações, como uma forma de melhorar o projeto existente sem alterar seu comportamento externo observável. Refatoração é o aspecto-chave para um projeto evolutivo, e é na evolução que se encontra a verdadeira sabedoria. Nesse caso, desenvolvimento ágil, evolutivo, incremental, apoiado em refatorações e testes, é uma excelente receita para garantir o sucesso do projeto, seja ela para um startup ou uma grande corporação.

Listagem 6. Classe Cliente após a refatoração Extract Method

```
unit GPauli.Sistemas.Locadora.Framework.Cliente;
interface
…
type
  Cliente = class
    private _nome: string;
    private _locacoes: TList<Locacao>;
    public function getNome(): string;
    public constructor Create(nome: string);
    public destructor Destroy(); override;
    public procedure addLocacao(locacao: Locacao);
    public function gerarRelatorioLocacao(): string;
    public function calculaPontos(each: Locacao): double;
  end;
implementation
{ Cliente }
…
function Cliente.gerarRelatorioLocacao(): string;
var
  quantidadeTotal: double;
  pontosLocacao: integer;
  total: double;
  each: Locacao;
begin
  quantidadeTotal := 0;
  pontosLocacao := 0;
  result := 'Relatório de locação para cliente ' + getNome() + sLineBreak;
  for each in _locacoes do
  begin
    total := calculaPontos(each);
    // adiciona ponto por locação
    pontosLocacao := pontosLocacao + 1;
    // adiciona bônus para locação de lançamento por mais de dois dias
    if ((each.getFilme().getCodigoPreco() = Filme.Lancamento) and
      (each.getDiasLocados() > 1)) then
        pontosLocacao := pontosLocacao + 1;
    // mostra filmes locados
    result := result + 'Filme: ' + each.getFilme().getTitulo() +
      ' - ' + FloatToStr(total) + ' pontos' + sLineBreak;
    quantidadeTotal := quantidadeTotal + total;
  end;
  // adiciona rodapé
  result := result + 'Total acumulado é ' +
    quantidadeTotal.ToString() + ' pontos ' + sLineBreak;
    result := result + 'Você ganhou ' + pontosLocacao.ToString() +
    ' pontos pela locação';
end;
function Cliente.calculaPontos(each: Locacao): double;
var
  total: double;
begin
  // determina total para cada locação
  case (each.getFilme.getCodigoPreco) of
    0:
      //Filme.Catalogo
      begin
        total := total + 2;
        if (each.getDiasLocados > 2) then
          total := total + (each.getDiasLocados - 2) * 1.5;
      end;
    1:
      //Filme.Lancamento
      begin
        total := total + each.getDiasLocados * 3;
      end;
    2:
      //Filme.Infantil
      begin
        if (each.getDiasLocados > 3) then
          total := total + (each.getDiasLocados - 3) * 1.5;
      end;
  end;
  result := total;
end;
end.
```

Refatoração	Finalidade
Extract Method	Extract Method é uma das refatorações mais comuns. Serve para compartilhar código comum evitando redundâncias. Serve também para quebrar grandes métodos em pequenas implementações. Ao extrair métodos, utiliza-se uma boa nomenclatura para representar sua intenção e evitar que outros códigos possam duplicar sua funcionalidade, evitando retrabalho.
Replace Temp with Query	Como variáveis temporárias são locais, elas podem encorajar a escrita de métodos muito longos, pois existem somente dentro do escopo do método onde são escritas. A substituição dessa variável por uma consulta permite que qualquer método na classe possa obter a informação desejada. A extração de variáveis para métodos de consulta é frequentemente realizada antes da refatoração Extract Method.
Move Method	Essa refatoração tem o propósito de mover métodos de uma classe que está realizando muitas atividades, repassando assim responsabilidades para outras classes e reduzindo acoplamento. Normalmente é usada para deslocar um método que consulta outro objeto, local onde deveria ser declarado de fato.
Self Encapsulate Field	Atributos podem ser acessados basicamente de duas formas. Quando estão dentro da mesma classe, eles podem ser acessos livremente. Quando acessados a partir de outras classes, eles devem ser encapsulados. A vantagem de aplicar esta refatoração para encapsular um atributo (campo) é permitir um controle maior sobre como os dados são acessados, podendo ainda fornecer um comportamento diferente para subclasses.
Replace Type Code with State / Strategy	Esta refatoração é semelhante à refatoração anterior, porém é aplicada quando não se pode criar subclasses, e ainda é necessário ter um comportamento específico para cada tipo. Neste caso pode-se utilizar um padrão de projeto, que pode ser o State ou Strategy, que são bastante similares. Esta refatoração consiste em criar uma classe para representar o tipo (objeto state), com suas respectivas subclasses. Onde originalmente existia o tipo do código, agora há uma referência para o objeto de estado, que pode assumir uma implementação específica (subclasse).
Replace Conditional with Polymorphism	Quando diferentes operações precisam ser realizadas com base no tipo de um objeto, normalmente utilizam-se ifs para testar cada possibilidade e assim aplicar cada uma das respectivas operações. São vários os problemas nessa abordagem. Primeiro, não é uma boa prática testar cada um dos tipos, pois a inclusão de um novo tipo exigirá a inclusão do mesmo no teste, modificando código que pode introduzir bugs. Segundo, se o teste for necessário em vários locais, haverá uma redundância de código. A solução proposta por esta refatoração é eliminar todo o teste condicional, deixando o mesmo por conta do compilador. A técnica consiste em criar uma classe base e subclasses para representar cada um dos tipos, e através de polimorfismo, invocar a operação que em tempo de execução tomará um comportamento diferente.
Rename Method	Métodos devem claramente comunicar sua intenção. Métodos bem nomeados dispensam comentários, no sentido literal da palavra. Esta refatoração quando aplicada permite renomear métodos para conter nomes mais sugestivos.

Tabela 1. Principais refatoração usadas no exemplo do artigo

Extraindo Métodos

No exemplo apresentado, o código que calcula a pontuação foi agora extraído para um novo método, chamado CalculaPontos. No local onde estava a sua implementação existe agora uma chamada ao novo método (**Listagem 6**).

Neste caso, foi aplicada uma refatoração muito comum chamada Extract Method, que basicamente ajuda a decompor um grande método em vários métodos menores, e fazer a chamada a eles nos locais originais onde estavam. Esta refatoração é a primeira que ajuda a resolver um clássico bad smell, chamado "Classe Deus" (God Class). Uma classe "Deus" é aquela que faz tudo, está muito sobrecarregada de responsabilidades, faz mais daquilo que inicialmente foi projetada para fazer. É impossível de reutilizar seu código, pois tudo nela depende dela mesmo (por exemplo, variáveis locais ou atributos). O seu índice de manutenção está comprometido, ou ainda sua complexidade ciclomática. Normalmente após um Extract Method, podemos aplicar um Move Method e retirar uma responsabilidade da classe para deixá-la mais simples, sem sobrecarga de responsabilidades. A **Tabela 1** mostra as principais refatoração utilizadas no exemplo deste artigo.

Algumas coisas fazem os programas difíceis de serem trabalhados, por exemplo: programas difíceis de ler são difíceis de modificar; programas que têm a lógica duplicada são difíceis de modificar; programas que requerem comportamento adicional que precisam modificar seu código de execução são difíceis de alterar; programas com lógica complexa são difíceis de modificar. Programas devem ser fáceis de ler, devem ter a lógica especificada em apenas um único lugar, que permitam ser modificadas sem alterar o comportamento externo observável, e que permita que lógicas condicionais possam ser expressas de forma simples.

Nota

Procure manter seus métodos com no máximo dez linhas de implementação, ou se for possível, cinco linhas, como sugerem alguns autores.

Se refatorações são importantes, porém não modificam o comportamento externo observável, como explicar a sua necessidade para gerentes, que normalmente não têm experiência técnica? Qualidade é um aspecto chave, este termo pode convencer a maioria dos gerentes do valor das revisões do código. A refatoração tende a quebrar grandes problemas em problemas menores (por exemplo, objetos). Porém, mais objetos precisam ser gerenciados. E objetos nesse caso normalmente costumam delegar tarefas a outros objetos, indiretamente. Por exemplo, ao colocar a lógica de uma tarefa em um método chamado por vários objetos, utiliza-se uma boa prática de compartilhamento. Além disso, a intenção do método e sua implementação são feitas separadamente.

Com isso, isola-se algo que pode mudar, evitando replicação ao mesmo tempo em que deixa o código limpo.

Movendo Métodos

O método extraído está agora implementado na classe Cliente. Porém, note que a mesma usa basicamente informações sobre locações, e não sobre clientes. Este método não deveria estar nesta classe, logo, deve ser movido para a classe de locação, através da refatoração chamada Move Method, como mostra a **Listagem 7.**

Listagem 7. Classe Filme após a refatoração Move Method

```
unit GPauli.Sistemas.Locadora.Framework.Locacao;
interface
uses
  GPauli.Sistemas.Locadora.Framework.Filme;
type
  Locacao = class
    private _filme: Filme;
    private _diasLocados: integer;
    public function calculaPontuacao(): double;
    public function getFilme(): Filme;
    public function getDiasLocados(): integer;
    public constructor Create(filme: Filme; diasLocados: integer);
  end;
implementation
{ Locacao }
...
function Locacao.calculaPontuacao(): double;
var
  total: double;
begin
  // determina total para cada locação
  case (getFilme().getCodigoPreco()) of
    0:
      //Filme.Catalogo
      begin
        total := total + 2;
        if (getDiasLocados() > 2) then
          total := total + (getDiasLocados() - 2) * 1.5;
      end;
    1:
      //Filme.Lancamento
      begin
        total := total + getDiasLocados() * 3;
      end;
    2:
      //Filme.Infantil
      begin
        if (getDiasLocados() > 3) then
          total := total + (getDiasLocados() - 3) * 1.5;
      end;
  end;
  result := total;
end;
end.
```

Agora a chamada ao antigo método na classe Cliente simplesmente delega a chamada para o novo método (**Listagem 8**), movido para a classe de locação.

Com isso estamos usando um princípio muito bom da programação orientada a objetos, a de separação de responsabilidades. Existe uma boa prática chamada SRP (Single Responsability Principle – Princípio da Responsabilidade Única), que diz que uma classe deve fazer somente uma coisa, ou seja, ter somente uma responsabilidade. Certamente o cálculo da pontuação não é uma responsabilidade da classe Cliente, não deveria estar implementado ali. Imagine, por exemplo, que a classe cliente precise ser agora reutilizada em um sistema de controle de estoque. Certamente não existe lá a necessidade de calcular pontos para uma locação.

O próximo passo é localizar todas as chamadas a este método e realizar a substituição para usar o novo método (**Listagem 9**).

Listagem 8. Método apenas delega cálculo para a classe de locação

```
function Cliente.calculaPontuacao(loc: Locacao): double;
begin
  result := loc.calculaPontuacao();
end;
```

Listagem 9. Classe Cliente após remoção de método de cálculo

```
function Cliente.gerarRelatorioLocacao(): string;
var
  quantidadeTotal: double;
  pontosLocacao: integer;
  total: double;
  each: Locacao;
begin
  quantidadeTotal := 0;
  pontosLocacao := 0;
  result := 'Relatório de locação para cliente ' + getNome() + sLineBreak;
  for each in _locacoes do
  begin
    total := each.calculaPontuacao();
    // adiciona ponto por locação
    pontosLocacao := pontosLocacao + 1;
    // adiciona bônus para locação de lançamento por mais de dois dias
    if ((each.getFilme().getCodigoPreco() = Filme.Lancamento) and
      (each.getDiasLocados() > 1)) then
        pontosLocacao := pontosLocacao + 1;
    // mostra filmes locados
    result := result + 'Filme: ' + each.getFilme().getTitulo() +
    ' -' + FloatToStr(total) + ' pontos' + sLineBreak;
    quantidadeTotal := quantidadeTotal + total;
  end;
  // adiciona rodapé
  result := result + 'Total acumulado é ' +
    quantidadeTotal.ToString() + ' pontos ' + sLineBreak;
  result := result + 'Você ganhou ' + pontosLocacao.ToString() +
  ' pontos pela locação';
end;
```

O método pode agora ser removido, o programa compila sem problema e os testes rodam normalmente. Veja que antes de remover o método, fizemos uma delegação local, ou seja, repassamos a chamada do cálculo para um objeto externo à classe. Isso faz com que apliquemos as refatorações em pequenas partes, primeiro isole o problema, recompile e teste, depois elimine, recompile e teste novamente. A delegação é uma interessante técnica da programação orientada a objetos, que aqui ajudou a facilitar a reestruturação interna do sistema de software. Sempre comece a reestruturação por estas principais refatorações, como Extract Method e Move Method. Elas ajudam a organizar melhoras as responsabilidades, dividir problemas grandes em problemas menores, maximizar a reutilização e leitura do código. Veja ainda que, com o apoio do

IDE do Delphi XE 5, o mesmo já identifica se uma variável ou objeto sairá do escopo do corpo do método, e cuida de passar os mesmos como parâmetros para o novo método extraído.

Eliminando variáveis temporárias

O próximo passo é eliminar a redundância ocasionada pelo uso da variável local chamada total. Ela simplesmente é configurada para receber o valor do cálculo de pontuação e não é modificada, logo pode ser eliminada usando a refatoração Replace Temp with Query, como mostra a **Listagem 10**.

Listagem 10. Classe Cliente após a refatoração Replace Temp with Query

```
function Cliente.gerarRelatorioLocacao(): string;
var
 quantidadeTotal: double;
 pontosLocacao: integer;
 each: Locacao;
begin
 quantidadeTotal := 0;
 pontosLocacao := 0;
 result := 'Relatório de locação para cliente ' + getNome() + sLineBreak;
 for each in _locacoes do
 begin
   // adiciona ponto por locação
   pontosLocacao := pontosLocacao + 1;
   // adiciona bônus para locação de lançamento por mais de dois dias
   if ((each.getFilme().getCodigoPreco() = Filme.Lancamento) and
     (each.getDiasLocados() > 1)) then
       pontosLocacao := pontosLocacao + 1;
   // mostra filmes locados
   result := result + 'Filme: ' + each.getFilme().getTitulo() +
   ' -' + FloatToStr(each.calculaPontuacao()) + ' pontos' + sLineBreak;
   quantidadeTotal := quantidadeTotal + each.calculaPontuacao();
 end;
 // adiciona rodapé
 result := result + 'Total acumulado é ' +
  quantidadeTotal.ToString() + ' pontos ' + sLineBreak;
 result := result + 'Você ganhou ' + pontosLocacao.ToString() +
 ' pontos pela locação';
end;
```

Variáveis temporárias são locais e podem forçar a escrita de métodos muito longos, pois existem somente dentro do escopo do método onde são escritas. Se ao declarar um método e em sua sessão var já inserir cinco variáveis locais, certamente esse método terá no mínimo 10 a 20 linhas de código. A substituição dessas variáveis por um método de consulta permite que qualquer método na classe possa obter a informação desejada. Basicamente, ao invés de criar uma variável local, cria-se uma função de escopo da classe e reduz o tamanho do método original. A extração de variáveis para métodos de consulta é frequentemente realizada antes da refatoração Extract Method.

Procedimento semelhante deve ser feito para pontosLocacao, como sendo o próximo passo da refatoração. Olhando para as variáveis locais, pontosLocacao é temporária, como mostra a **Listagem 11**. Já a variável each usada nesse caso pode ser passada como parâmetro para um novo método, como mostra a **Listagem 12**.

Variáveis temporárias podem causar problemas, a menos que sejam usadas somente dentro de suas próprias rotinas.

No exemplo existem duas variáveis temporárias, ambas usadas para obter um valor total a partir das locações de um determinado cliente. As implementações da geração dos relatórios de locação, tanto em texto quanto para Web, requerem esse total. Nesse caso pode ser utilizada a refatoração Replace Temp With Query para substituir quantidadeTotal e pontosLocacao por métodos de consulta (query). Essas consultas são acessíveis a qualquer método na classe e podem encorajar a criar um bom design sem métodos longos ou complexos, o que normalmente ocorre quando se utilizam variáveis temporárias. Essas refatorações podem ser vistas nas **Listagens 13** a **16**.

Listagem 11. Classe Cliente após a refatoração Extract Method e Move Method

```
function Cliente.gerarRelatorioLocacao(): string;
var
 quantidadeTotal: double;
 pontosLocacao: integer;
 each: Locacao;
begin
 quantidadeTotal := 0;
 pontosLocacao := 0;
 result := 'Relatório de locação para cliente ' + getNome() + sLineBreak;
 for each in _locacoes do
 begin
   // adiciona ponto por locação
   pontosLocacao := each.calculaPontosLocacao();
   // mostra filmes locados
   result := result + 'Filme: ' + each.getFilme().getTitulo() + ' -' + FloatToStr
   (each.calculaPontuacao()) + ' pontos' + sLineBreak;
   quantidadeTotal := quantidadeTotal + each.calculaPontuacao();
 end;
 // adiciona rodapé
 result := result + 'Total acumulado é ' +
  quantidadeTotal.ToString() + ' pontos ' + sLineBreak;
 result := result + 'Você ganhou ' + pontosLocacao.ToString() + ' pontos pela locação';
end;
```

Listagem 12. Classe Locacao após a refatoração Move Method

```
function Locacao.calculaPontosLocacao(): integer;
begin
 // adiciona bônus para locação de lançamento por mais de dois dias
  if ((getFilme().getCodigoPreco() = Filme.Lancamento) and
     (getDiasLocados() > 1)) then
       result := 2
     else
       result := 1;
end;
```

Listagem 13. Classe Cliente após a refatoração Replace Temp with Query

```
function Cliente.gerarRelatorioLocacao(): string;
var
 pontosLocacao: integer;
 each: Locacao;
begin
 pontosLocacao := 0;
 result := 'Relatório de locação para cliente ' + getNome() + sLineBreak;
 for each in _locacoes do
 begin
   // adiciona ponto por locação
   pontosLocacao := each.calculaPontosLocacao();
   // mostra filmes locados
   result := result + 'Filme: ' + each.getFilme().getTitulo() +
   ' -' + FloatToStr(each.calculaPontuacao()) + ' pontos' + sLineBreak;
 end;
 // adiciona rodapé
 result := result + 'Total acumulado é ' +
  getQuantidadeTotal().ToString() + ' pontos ' + sLineBreak;
 result := result + 'Você ganhou ' + pontosLocacao.ToString() + ' pontos pela locação';
end;
```

Listagem 14. Novo método gerado na classe Cliente

```
function Cliente.getQuantidadeTotal(): double;
var
 each: Locacao;
begin
 result := 0;
 for each in self._locacoes do
  result := result + each.calculaPontuacao();
end;
```

Listagem 15. Classe Cliente após outra refatoração Replace Temp with Query

```
function Cliente.gerarRelatorioLocacao(): string;
var
 each: Locacao;
begin
 result := 'Relatório de locação para cliente ' + getNome() + sLineBreak;
 for each in _locacoes do
 begin
   // mostra filmes locados
   result := result + 'Filme: ' + each.getFilme().getTitulo() +
    ' - ' + FloatToStr(each.calculaPontuacao()) + ' pontos' + sLineBreak;
 end;
 // adiciona rodapé
 result := result + 'Total acumulado é ' +
  getQuantidadeTotal().ToString() + ' pontos ' + sLineBreak;
 result := result + 'Você ganhou ' + getPontosLocacao().ToString() +
' pontos pela locação';
end;
```

Listagem 16. Mais um método gerado na classe Cliente

```
function Cliente.getPontosLocacao(): integer;
var
 each: Locacao;
begin
 result := 0;
 for each in self._locacoes do
  result := result + each.calculaPontosLocacao()
end;
```

Normalmente, uma refatoração diminui a quantidade de código, porém não foi o caso aqui, isso será resolvido adiante. Outra observação é com relação à performance, visto que alguns programadores escolhem não escrever métodos pequenos porque eles temem os custos de desempenho associados a encadear chamadas para muitos pequenos métodos. Essa é uma escolha errada, pois bons projetistas não otimizam código prematuramente. Encadear chamadas a métodos pequenos normalmente custa pouco em termos de desempenho. E nada impede que uma refatoração futura resolva estes problemas de desempenho. Além disso, o próprio Delphi oferece suporte para geração de métodos realinhados usando a palavra-reservada inline. Ou seja, no fonte o método estará extraído, porém ao compilar, o Delphi desfaz a chamada e recoloca o código no local original, isso afeta somente o código compilado (EXE, DLL, DCU), não o código-fonte, que é o alvo da refatoração. Isso evita problemas de performance e criação de call stacks muito grandes em memória. Nesse caso, o código consta então em um método pequeno apenas para tornar o design mais elegante, porém o compilador irá gerar um código otimizado, como se ele não existisse no código fonte de origem.

Essas consultas estão agora disponíveis para qualquer código escrito dentro da classe Cliente. Elas podem ser expostas na interface da classe para que outras partes do sistema possam acessar. Sem essas consultas, outros métodos precisariam realizar os laços, exigindo mais código que seria difícil de escrever e manter. Neste ponto é possível parar com a refatoração para incluir a nova funcionalidade, a geração de declaração em HTML (**Listagem 17**) e os apropriados testes.

Listagem 17. Gerando relatório de locação em formato HTML

```
function Cliente.gerarRelatorioLocacaoHTML(): string;
var
 each: Locacao;
begin
 result := '<H1>Relatório de locação para cliente <EM>'
 + getNome() + '</EM></H1>' + sLineBreak;
 for each in _locacoes do
 begin
   // mostra filmes locados
   result := result + 'Filme: ' + each.getFilme().getTitulo() +
    ' - ' + FloatToStr(each.calculaPontuacao())
     + ' pontos <BR>' + sLineBreak;
 end;
 // adiciona rodapé
 result := result + '<p>Total acumulado é <EM>' +
  getQuantidadeTotal().ToString() + ' pontos </EM></p>' + sLineBreak;
 result := result + '<p>Você ganhou <EM>'
 + getPontosLocacao().ToString() +
' pontos </EM> pela locação</p>';
end;
```

Veja que essa é uma boa oportunidade para refatorar, ou seja, quando uma nova feature deve ser adicionada. A refatoração nesse caso pode ajudar a tornar o código mais fácil de entender imediatamente. Melhorando o projeto, a nova funcionalidade pode ser adicionada de forma mais simples e rápida, sem introduzir novos bugs. A refatoração vai ajudar a encontrar bugs mais facilmente, porque o código se torna mais legível, mais claro. Outro bom momento para refatorar é quando são feitas revisões no código. Grandes empresas realizam constantes revisões no código a fim de garantir a qualidade, o que ajuda a na transmissão conhecimento dos mais experientes aos menos experientes, pois ajudam a sugerir ideias. As refatorações nesse caso são uma forma mais concreta de se obter resultados de uma revisão de código.

No exemplo, agora todo o código relativo a cálculos foi reaproveitado em ambos os métodos, ele não está duplicado. Se as regras destes cálculos mudarem, um único local do código precisa ser alterado. Ainda existe um problema no código, mais especificamente na classe de locação, onde é feito o teste condicional com case do Pascal. Não é uma boa ideia usar este tipo de teste baseado nos atributos de outros objetos, os dados devem ser do próprio objeto. Isso implica em mover o método para a classe Filme (**Listagem 18**).

O número de diárias foi passado como parâmetro, proveniente da classe de locação. O método usa basicamente duas informações, o tipo de filme e o período. Note que foi passado o número de dias e não o tipo de filme, pois provavelmente novos tipos podem

ser adicionados. Informações de tipos normalmente tendem a sofrer modificações, e uma mudança destas provavelmente exigiria mudanças em várias outras partes do código. O mesmo foi feito para o cálculo da pontuação (**Listagem 19**). Isso mantém as possíveis mudanças que dependem de um tipo dentro da própria classe que define o tipo.

Listagem 18. Método movido para a classe Filme

```
function Filme.calculaPontuacao(diasLocados: integer): double;
var
 total: double;
begin
 // determina total para cada locação
 case (getCodigoPreco()) of
  0: // Filme.Catalogo
   begin
    total := total + 2;
    if (diasLocados > 2) then
     total := total + (diasLocados - 2) * 1.5;
   end;
  1: // Filme.Lancamento
   begin
    total := total + diasLocados * 3;
   end;
  2: // Filme.Infantil
   begin
    if (diasLocados > 3) then
     total := total + (diasLocados - 3) * 1.5;
   end;
 end;
 result := total;
end;
```

Listagem 19. Método movido para a classe Filme

```
function Filme.calculaPontosLocacao(diasLocados: integer): integer;
begin
 // adiciona bônus para locação de lançamento por mais de dois dias
 if ((getCodigoPreco() = Filme.Lancamento) and
    (diasLocados > 1)) then
     result := 2
    else
     result := 1;
end;
```

Aplicando polimorfismo e refatoração para Padrões de Projeto

Padrões de Projeto ajudam a tornar sistemas de software mais fáceis de serem mantidos e evoluídos. Eles são constantemente modificados e adaptados, novos recursos são implementados, bugs são corrigidos, outros criados e aparecem do nada, novas exigências são feitas, o código vai se tornando mais complexo e se afasta do seu projeto original, diminuindo sua qualidade. Se uma aplicação precisa ser modificada em um determinado momento, seja para manutenção corretiva ou evolutiva para atender novas funcionalidades, este processo precisa ser feito de forma disciplinada, correta, usando padrões, já que o custo da manutenção representa um porcentual elevado comparado ao custo total do sistema, como citado. Com isso, o uso de padrões de projeto, ou Design Patterns, torna um software mais fácil de ser mantido em longo prazo, reduzindo custos ou otimizando o retorno do investimento, apesar de exigir um maior esforço em um primeiro momento, durante o design do projeto a ser realizado. No entanto, aplicar padrões antecipadamente em um projeto requer mais tempo e aumenta custos, ao mesmo tempo em que aumenta sua complexidade, não se pode aplicar padrões antecipadamente para prever essas mudanças.

No exemplo apresentado, existem no código três tipos de filmes que têm diferentes maneiras de responder a mesma questão, o que sugere a criação de uma hierarquia de classes através do uso de herança, conforme mostra a **Figura 3**. Esse tipo de estrutura permite substituir a instrução condicional por polimorfismo. Porém existe um pequeno detalhe, um filme pode mudar sua classificação um dia (um lançamento um dia não será mais lançamento), porém uma classe não perde sua herança durante a sua "vida". Existe uma solução já conhecida para este problema, um padrão de projeto chamado State. A solução é apresentada na **Figura 4**.

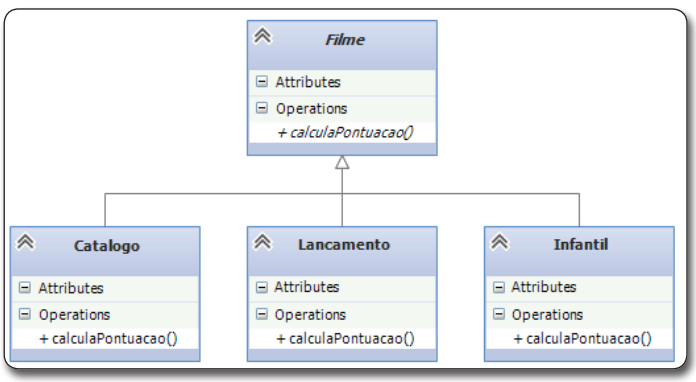


Figura 3. Hierarquia de classes, com subclasses para cada tipo de filme

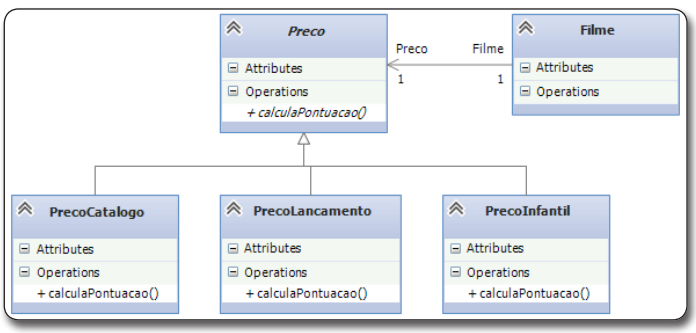


Figura 4. Aplicando o padrão State

A intenção do padrão State é permitir que um objeto altere seu comportamento quando seu estado interno muda.

O padrão pode ser aplicado quando: i) o comportamento de um objeto depende do seu estado e ele pode mudar seu em tempo de execução, dependendo desse estado; ii) operações têm comandos condicionais grandes, de várias alternativas, que dependem do estado do objeto. Frequentemente, várias operações conterão essa mesma estrutura condicional. O padrão State coloca cada ramo do comando adicional em uma classe separada. Isto lhe permite tratar o estado do objeto como um objeto propriamente dito, que pode variar independentemente de outros objetos. Entre alguns benefícios podemos citar:

1) o padrão confina comportamento específico de estados e particiona o comportamento para estados diferentes;
2) ele torna explícitas as transações de estado;
3) objetos state podem ser compartilhados.

O State basicamente é uma delegação polimórfica de um condicional resolvida por um padrão Strategy. Normalmente o problema poderia ser resolvido pela simples aplicação de polimorfismo, e a seguir, do padrão Strategy. Como o estado do objeto pode afetar seu comportamento, simplesmente se delega a cadeia polimórfica para uma família externa, e por associação, e consequentemente delegação interna, se resolve o algoritmo que é processado externamente à classe. State e Strategy são padrões muito parecido e têm intenções e implementações semelhantes, e ajudam a resolver problemas relacionados ao uso excessivo de condicionais para escolher um determinado algoritmo, o que compromete a complexidade ciclomática, a cobertura de testes unitários e ainda tende a duplicar código. A diferença básica é que no State o comportamento polimórfico fica separado do tratamento de estado.

Para aplicar este padrão, foram aplicadas três refatorações: Replace Type Code with State/Strategy, Move Method e Replace Conditional with Polymorphism. Para aplicar a primeira refatoração, é necessário aplicar Self Encapsulate Field para garantir que todos os acessos aos tipos passem por métodos get e set (**Listagem 20**).

Listagem 20. Aplicando encapsulamento em campo

```
Filme = class
   private _codigoPreco: integer;
   public function getCodigoPreco(): integer;
   public procedure setCodigoPreco(codigoPreco: integer);
   public property CodigoPreco: integer
                read getCodigoPreco write setCodigoPreco;

...
function Filme.getCodigoPreco(): integer;
begin
  result := self._codigoPreco;
end;
procedure Filme.setCodigoPreco(codigoPreco: integer);
begin
  self._codigoPreco := codigopreco;
end;
constructor Filme.Create(titulo: string; codigoPreco: integer);
begin
  self._titulo := titulo;
  setCodigoPreco(codigoPreco);
end;
```

Após compilar e testar tudo continua funcionando corretamente. Neste ponto é possível adicionar as novas classes. A classe base define um método abstrato que é então sobrescrito nas classes concretas descendentes (**Listagem 21**). Veja que neste caso, a abstração é a capacidade de representar conceitos do domínio do problema, ressaltando apenas o que for relevante para a aplicação em questão. A definição da classe em Delphi atende a esse requisito, pois ela é um modelo limitado de alguma entidade real, que implementa apenas as características que são necessárias para a aplicação sendo desenvolvida em um dado contexto. Já o polimorfismo é a capacidade de classes descendentes podem implementar de forma diferente um conjunto de operações comuns. Polimorfismo é a peça-chave para a aplicação de refatoração para os padrões de projeto Strategy e State, entre outros.

Nota

A refatoração, no entanto, também tem seus problemas. Muitas aplicações de banco de dados são fortemente acopladas com esquemas de banco de dados. Uma solução é considerar a utilização de uma camada intermediária de comunicação com o banco de dados, que isole muitas das responsabilidades, permitindo flexibilidade. Outro problema diz respeito à modificação de interfaces de objetos. Bons projetistas modelam classes (interfaces) e relações, de forma que uma modificação possa ser feita internamente na implementação sem modificar suas interfaces públicas. Porém, muitas refatorações modificam essas interfaces, por exemplo, a Rename Method. Isso desencadeará uma série de modificações em todos os locais onde o método é invocado. Uma boa solução é manter uma versão de transição, fazendo que a interface antiga ainda possa ser usada até que a nova a substitua completamente. Uma boa dica é nunca publicar interfaces prematuramente, pois interfaces, segundo os fundamentos da programação orientada a objetos, devem (ou deveriam) ser imutáveis.

Listagem 21. Método movido para a classe Filme

```
unit GPauli.Sistemas.Locadora.Framework.Preco;
interface
type
  Preco = class abstract
    public function getCodigoPreco(): integer; virtual; abstract;
  end;
  PrecoLancamento = class(Preco)
    public function getCodigoPreco(): integer; override;
  end;
  PrecoCatalogo = class(Preco)
    public function getCodigoPreco(): integer; override;
  end;
  PrecoInfantil = class(Preco)
    public function getCodigoPreco(): integer; override;
  end;
implementation
uses
  GPauli.Sistemas.Locadora.Framework.Filme;
{ PrecoInfantil }
function PrecoInfantil.getCodigoPreco(): integer;
begin
  result := Filme.Infantil;
end;
{ PrecoCatalogo }
function PrecoCatalogo.getCodigoPreco(): integer;
begin
  result := Filme.Catalogo;
end;
{ PrecoLancamento }
function PrecoLancamento.getCodigoPreco(): integer;
begin
  result := Filme.Lancamento;
end;
end.
```

É necessário modificar os métodos assessores do preço da classe Filme para usarem estas novas classes. Nesse ponto é necessário substituir o código que representa o preço pelo tipo do preço (**Listagem 22**). Agora é possível aplicar o Move Method para o cálculo de pontos (**Listagem 23**).

Listagem 22. Organizando os assessores

```
Filme = class
  private _preco: Preco;
  private _codigoPreco: integer;
  public function getCodigoPreco(): integer;
  public procedure setCodigoPreco(codigoPreco: integer);
...
function Filme.getCodigoPreco(): integer;
begin
 result := self._codigoPreco;
end;
procedure Filme.setCodigoPreco(codigoPreco: integer);
begin
 case codigoPreco of
  0: self._preco := PrecoCatalogo.Create();
  1: self._preco := PrecoLancamento.Create();
  2: self._preco := PrecoInfantil.Create();
 end;
end;
```

Listagem 23. Método movido para a classe abstrata Preco

```
function Preco.calculaPontuacao(diasLocados: integer): double;
var
 total: double;
begin
 // determina total para cada locação
 case (getCodigoPreco()) of
  0: // Filme.Catalogo
   begin
    total := total + 2;
    if (diasLocados > 2) then
     total := total + (diasLocados - 2) * 1.5;
   end;
  1: // Filme.Lancamento
   begin
    total := total + diasLocados * 3;
   end;
  2: // Filme.Infantil
   begin
    if (diasLocados > 3) then
     total := total + (diasLocados - 3) * 1.5;
   end;
 end;
 result := total;
end;
```


CURSOS ONLINE

A Revista Java Magazine oferece aos seus assinantes uma série de Cursos Online de alto padrão de qualidade.

CONHEÇA ALGUNS DOS CURSOS:

- Curso de noSQL (Redis) com Java
- Curso Básico de JDBC
- Java Básico: Aplicações Desktop
- JSF com Primefaces
- Conhecendo o Apache Struts

Para mais informações :
www.devmedia.com.br/curso/javamagazine
(21) 3382-5038

DEVMEDIA

Listagem 24. Classes Preco, PrecoLancamento, PrecoCatalogo e PrecoInfantil

```
unit GPauli.Sistemas.Locadora.Framework.Preco;
interface
type
 Preco = class abstract
 private
  public function getCodigoPreco(diasLocados: integer): double; virtual;
abstract;
 end;
 PrecoLancamento = class(Preco)
  public function getCodigoPreco(diasLocados: integer): double; override;
 end;
 PrecoCatalogo = class(Preco)
  public function getCodigoPreco(diasLocados: integer): double; override;
 end;
 PrecoInfantil = class(Preco)
  public function getCodigoPreco(diasLocados: integer): double; override;
 end;
implementation
uses
 GPauli.Sistemas.Locadora.Framework.Filme;
{ PrecoInfantil }
function PrecoInfantil.getCodigoPreco(diasLocados: integer): double;
begin
 result := 1.5;
 if (diasLocados > 3) then
  result := (diasLocados - 3) * 1.5;
end;
{ PrecoCatalogo }
function PrecoCatalogo.getCodigoPreco(diasLocados: integer): double;
begin
 result := 2;
 if (diasLocados > 2) then
  result := result + (diasLocados - 2) * 1.5;
end;
{ PrecoLancamento }
function PrecoLancamento.getCodigoPreco(diasLocados: integer): double;
begin
 result := diasLocados * 3;
end;
end.
```

Listagem 25. Método movido para a classe Preco

```
 Preco = class abstract
 private
  public function calculaPontosLocacao(diasLocados: integer): integer; virtual;
...
function Preco.calculaPontosLocacao(diasLocados: integer): integer;
begin
 // adiciona bônus para locação de lançamento por mais de dois dias
 if ((getCodigoPreco(diasLocados) = Filme.Lancamento) and
     (diasLocados > 1)) then
      result := 2
     else
      result := 1;
end;
```

Os métodos assessores são amplamente utilizadas na programação orientada a objetos para promover o encapsulamento de um atributo. Basicamente, aquilo que era público passa a ser privado, e dois métodos (os famosos "get" e "set") tratam de promover a leitura e a escrita das informações no atributo interno, que normalmente é referenciada como a posição de memória que realmente guarda a informação na memória, ficando escondida, privada, encapsulada. Dessa forma, os assessores permitem o monitoramento das operações de escrita e leitura do atributo, podendo se fornecer um comportamento em resposta a uma dessas operações. No Delphi, o conceito de propriedade (property) e das palavras-reservadas read / write implementam assessores, ou pode-se implementar o comportamento de forma como no código, sem se explicitar uma propriedade pública. Esta é uma abordagem mais tradicional da orientação a objetos, que não usa o conceito de propriedades. Propriedades são utilizadas em linguagens como Delphi, C# e VB.NET, e basicamente são uma forma de se escrever métodos get / set usando uma estrutura própria.

Neste ponto pode-se aplicar a refatoração Replace Conditional by Polymorphism. As classes da **Listagem 24** contêm os códigos que foram removidos das condicionais para suas respectivas classes. Cada método deve conter a palavra-reservada **override** para indicar que está sobrescrevendo o método abstrato (abstract) da classe base através do uso de polimorfismo. Finalmente, o método que calcula os pontos é então movido para a classe Preco (**Listagem 25**). Neste caso o método não é abstrato na classe base, ele foi marcado como virtual para que possa oferecer um comportamento padrão, podendo ser sobrescrito com um novo comportamento em classes descendentes.

Esta etapa finaliza a refatoração do sistema inicial. Obviamente poderíamos aplicar ainda outras refatorações, especialmente as voltadas para alguns padrões de projeto. Veja que um bom contraste com refatoração é performance e otimização. Assim como refatoração, melhorar a performance e otimização de um sistema de software não afeta o comportamento observável. No entanto, têm propósitos diferentes, pois melhorar a performance pode tornar o código mais fácil de entender. Para o usuário final, um sistema de software refatorado terá as mesmas funções anteriores. A refatoração ajuda no processo de identificação de bugs, pois clarifica a estrutura do programa e torna mais fácil a localização de problemas. Refatoração ajuda a programar de forma mais rápida, apesar de isso parecer estranho. Melhorar o projeto e legibilidade reduz a velocidade do desenvolvimento, mas em longo prazo, esses fatores vão tornar mais fáceis as adições de novas funcionalidades e correções de bugs, que serão feitas de forma mais rápida. Como vimos, a refatoração não é uma atividade que deve ter um tempo programado para realizá-la, ela deve ser feita continuamente, em pequenos passos. Este exemplo serve como uma excelente base para conhecer

os princípios e vantagens da refatoração em pequenos passos, apoiadas por testes.

Veja que o fato de modificar o código existente, evoluindo-o para uma nova arquitetura, tornou mais simples a inclusão de uma nova funcionalidade, eliminando a redundância de código e isolando partes complexas ou que possam sofrer novas mudanças. Com isso se tem uma boa base para examinar os demais fundamentos envolvidos no estudo da refatoração.

Links:

Refactoring
www.refactoring.com

Autor

Guinther Pauli
Arquiteto de Software certificado pela Microsoft, Consultor, Instrutor, Autor, Palestrante, Professor de Graduação e de Pós-Graduação, Colaborador Editorial das Revistas .NET Magazine Brasil e ClubeDelphi, Certificado oficial Microsoft MCP, MCAD, MCSD.NET, MCTS, MCPD (ASP.NET, C# e Arquitetura) - Certificado oficial Delphi: 3, 5, 6, 7, 2005, 2006, Delphi para Web, Kylix. Most Valuable Professional (MVP).

Você gostou deste artigo?

Dê seu voto em ***www.devmedia.com.br/clubedelphi/feedback***

Ajude-nos a manter a qualidade da revista!

CURSOS ONLINE

A Revista SQL Magazine oferece aos seus assinantes uma série de Cursos Online de alto padrão de qualidade.

Feita para Desenvolvedores de Software e DBAs
SQL magazine

CONHEÇA OS CURSOS MAIS RECENTES:

- **Cursos: Curso de noSQL (Redis) com Java**
- **Desenvolvimento para SQL Server com .NET**
- **Curso PostgreSQL - Treinamento de banco de dados (Curso Básico)**

Para mais informações :
http://www.devmedia.com.br/cursos/banco-de-dados
(21) 3382-5038

DEVMEDIA

Atualização de arquivos em Delphi

Desenvolvimento de um Launcher

Ao se trabalhar com um número grande de clientes começamos a ter problemas de como controlar as versões previamente instaladas. Até certo número, a atualização manual disponibilizada através de um link para download ou até mesmo acessando os computadores dos clientes para fazer a atualização não é a melhor das soluções, mas ainda assim é possível. Porém, quando ocorre um aumento de clientes, executáveis e arquivos comuns à distribuição, tais como relatórios, DLLs, BPLs entre outras, o processo torna-se mais complexo e podemos então encontrar algumas dificuldades no processo de atualização. Nesse momento começam a surgir soluções de atualizações que em sua maioria, não são eficazes.

Muitos desenvolvedores utilizam o próprio build da versão como um processo de verificação, porém, devido à falta de automatização, esse arquivo pode vir a não ser devidamente atualizado e então o processo também torna-se falho. Outra situação comumente utilizada é a validação pela data de alteração do arquivo, porém, como sabe-se, facilmente podemos burlar este processo alterando a própria data do Sistema Operacional.

Uma solução para os problemas descritos é a utilização de comparação de MD5. Ao se alterar apenas um caractere no arquivo fonte, e sem a necessidade do build, o MD5 é alterado, garantindo assim a identificação de uma versão diferente no computador do cliente.

O que é MD5

O MD5 (Message-Digest algorithm 5) é um algoritmo de hash de 128 bits desenvolvido pela RSA Data Security Inc., e muito utilizado na verificação de integridade de arquivos.

Seu propósito de desenvolvimento foi suceder o modelo MD4 que possuía alguns problemas de segurança. A verificação é realizada pela comparação das duas hashes sendo uma a original e a outra à MD5 gerada.

Os hashes MD5 são representados por uma sequência de 32 caracteres hexadecimais. Mesmo uma pequena alteração, criará um hash completamente diferente.

Fique por Dentro

Muitos criam sistemas de atualização toda vez que lançam uma nova versão do software, alguns optam por fazer a atualização manual e em outros casos, disponibiliza-se em um servidor FTP e se repassa para os clientes o link para download. A correlação em todos esses métodos é que não existe a garantia que o sistema está atualizado. Com um mecanismo de Launcher de atualização, veremos como é possível garantir as versões dos executáveis e arquivos de nosso sistema antes da abertura do sistema.

Neste artigo veremos como validar as versões utilizando o modelo de criptografia MD5 consultando uma base de dados online.

Como é validada a estrutura de arquivos

Para realizar a validação da estrutura de arquivos, será criada uma tabela a ser utilizada como dicionário de estrutura, onde iremos consultar pelo sistema se existem atualizações dos arquivos. O Launcher será responsável por consultar o dicionário e comparar o MD5 do arquivo com o informado no dicionário, caso o MD5 do arquivo não seja igual ao informado no dicionário, esse será extraído e colocado no lugar do arquivo desatualizado. Para o devido funcionamento, essa base deve ficar disponível em um servidor na Web com alta disponibilidade e as máquinas clientes devem estar em um ambiente com acesso à internet para que sempre que o Launcher seja executado, essa consulta possa ser devidamente realizada.

O banco de dados

Para nossa solução, faremos o uso do banco de dados o MySQL. A **Figura 1** apresenta a modelagem proposta. A descrição de cada campo pode ser observada nas **Tabelas 1** e **2**.

O intuito de utilizar os campos de controle de versões, motivo de terem sido disponibilizadas, bem como o responsável é a possibilidade de elaboração de relatórios futuros, proporcionando com o passar do tempo uma base de conhecimento e logs das versões.

Uma forma de "obrigarmos" a aplicação a só abrir após a validação pelo Launcher é trabalharmos com parâmetros de inicialização. Neles definiremos que nossa aplicação só será aberta

se um determinado valor for passado. Com isso garantimos que a aplicação não irá ser aberta sem as devidas atualizações, evitando assim alguns problemas que ocorrem no cotidiano, como a comum divergência entre versões. Vale lembrar que para comunicar-se com o banco de dados, utilizaremos a suíte MyDac (ver **BOX 1**).

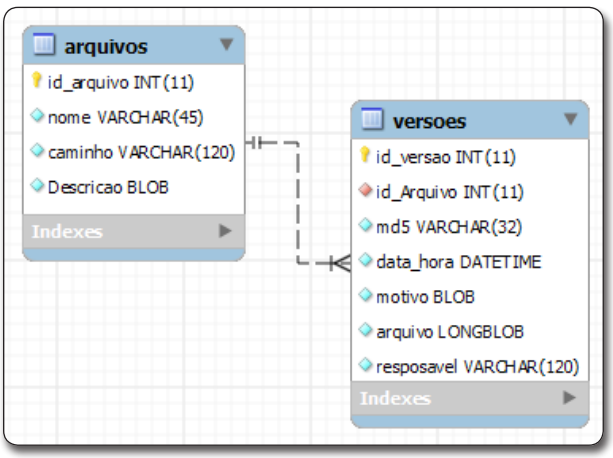


Figura 1. Estrutura de tabelas

Campo	Descrição
id_arquivo	Identificador da chave primária
nome	Nome do arquivo
caminho	Caminho onde deve-se buscar a máquina cliente
descricao	Descrição do arquivo

Tabela 1. Descrição da tabela ARQUIVOS

Campo	Descrição
id_versao	Identificador da chave primária
id_arquivo	Chave estrangeira com a tabela ARQUIVOS
md5	Hash correspondente ao arquivo
data_hora	Descrição do arquivo
motivo	Motivo de ser disponibilizada uma nova versão
arquivo	Campo responsável por armazenar o arquivo
responsavel	Pessoa responsável por ter disponibilizado a versão

Tabela 2. Campos da tabela de Versoes

BOX 1. MyDac

O MyDac é um conjunto de componentes de acesso nativo ao MySQL no Delphi. Aplicações desenvolvidas com o MyDac efetuam conexões direta com o banco de dados sem a necessidade de DLL cliente como o caso do Zeos e o DBExpress por exemplo. O MyDAC não requer instalação de drivers de conexão como ODBC.

Criando o banco de dados e sua estrutura

Primeiramente devemos criar uma base de dados, em que o nome sugerido será "dic_estrutura". Para criar as tabelas de fato, basta que no console do MySQL seja inserido o script da **Listagem 1.**

Listagem 1. Script para criar a estrutura de nossa base de dados no MySQL

```
01 -- Cria a base de dados
02 CREATE SCHEMA `dic_estrutura`;
03
04 -- Cria a tabela de arquivos
05 CREATE TABLE IF NOT EXISTS `dic_estrutura`.`Arquivos` (
06   `id_arquivo` INT              NOT NULL AUTO_INCREMENT,
07   `nome`       VARCHAR(45)  NOT NULL,
08   `caminho`    VARCHAR(120) NOT NULL,
09   `Descricao`  BLOB             NOT NULL,
10 PRIMARY KEY (`id_arquivo`))
11 ENGINE = InnoDB;
12
13 -- Cria a tabela de versões
14 CREATE TABLE IF NOT EXISTS `dic_estrutura`.`Versoes` (
15   `id_versao`   INT              NOT NULL AUTO_INCREMENT,
16   `id_Arquivo`  INT              NOT NULL,
17   `md5`         VARCHAR(32)      NOT NULL,
18   `data_hora`   DATETIME         NOT NULL,
19   `motivo`      BLOB             NOT NULL,
20   `arquivo`     LongBlob         NOT NULL,
21   `resposavel` VARCHAR(120)    NOT NULL,
22  PRIMARY KEY (`id_versao`),
23  INDEX `FK_id_Arquivo_idx` (`id_Arquivo` ASC),
24  CONSTRAINT `FK_id_Arquivo`
25  FOREIGN KEY (`id_Arquivo`)
26  REFERENCES `dic_estrutura`.`Arquivos` (`id_arquivo`)
27 )
28 ENGINE = InnoDB;
```

Upload dos arquivos

Com a estrutura de dados definida, vamos dar início ao nosso sistema de envio de arquivos. O aplicativo em questão não será responsável somente por enviar os arquivos para a base de dados, além disso, permitirá o cadastro de nossos arquivos e seus respectivos locais, preenchendo os dados necessários para que o Launcher efetue as consultas.

No formulário principal adicionaremos um componente de conexão com o banco de dados. Por se tratar de um exemplo pequeno e apenas em caráter de ensino, a conexão será alocada no formulário principal. Adicione um TMyConnection e configure o mesmo para acessar a base de dados anteriormente criada. Para acessar o banco de dados com TMyConnection, basta configurarmos as propriedades como apresentado na **Tabela 3**. Observe que o processo é o mesmo para as demais suítes de componentes de dados, como o SQLConnection do DBExpress por exemplo.

Propriedade	Valor	Descrição
Database	dic_estrutura	Nome da base de dados que o componente irá acessar
Server	Meu_Servidor*	Ip\Nome\DNS do servidor MySQL que será acessado
Username	Meu_Usuario*	Usuário do banco de dados
Password	Meu_Password*	Senha do usuário do banco de dados
LoginPrompt	False	Define se a tela de confirmação de senha (login) irá aparecer a cada nova conexão
Connected	True	Conecta-se ao banco de dados

Tabela 3. Propriedades e configurações do componente TMyConnection

Nota

Substituir os valores com o caractere * por valores reais.

Após configurarmos o componente de conexão, vamos elaborar o formulário principal da aplicação de envio de arquivos. A **Figura 2** temos uma apresentação de como ficou o layout proposto para o artigo.

Além do formulário exibido, vamos adicionar mais dois na aplicação. Eles terão as funcionalidades de adicionar e enviar arquivos. Primeiramente crie um novo formulário e alteres seu Caption para "Adicionar Arquivo", salvando-o com o nome que desejar. Em seguida, acrescente o formulário principal ao seu uses, inserindo em seguida um componente TMyQuery. Renomeie-o para "Qry_Arquivo" e informe em sua propriedade Connection o componente de conexão definido no form principal. A TMyQuery para que já trabalhou com Interbase Express (IBX), assemelha-se por utilizar a guia de criação das instruções de atualização por meio do componente. Para fazer uso desses recursos basta realizar um duplo clique sobre o mesmo e será apresentada uma tela semelhante a da **Figura 3**.

Figura 2. Formulário principal do sistema de envio de arquivos

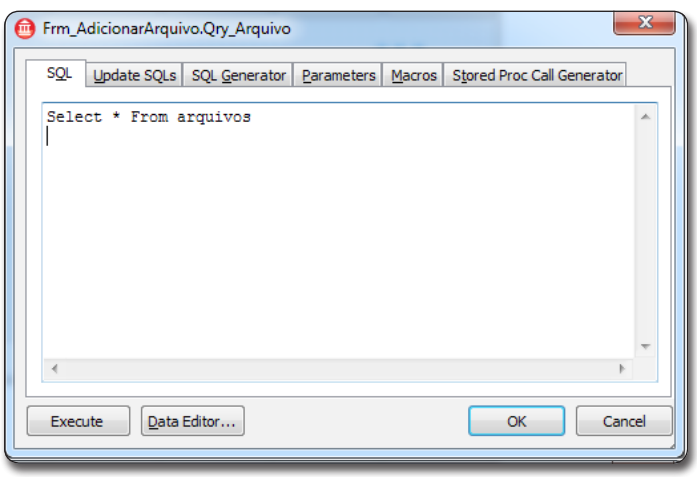


Figura 3. Formulário de definição de CRUD para o componente TMyQuery

Na janela que se abre (SQL), vamos inserir a instrução de consulta a tabela de arquivos, na aba SQL_Generator, bastando clicar em "Generate SQL" para que nossos scripts de CRUD sejam automaticamente gerados. Para verificar ou alterar os scripts, basta abrir a aba "Update SQLs" e, agora com a query montada, basta acrescentar e configurar os componentes visuais, conforme mostra a **Figura 4**.

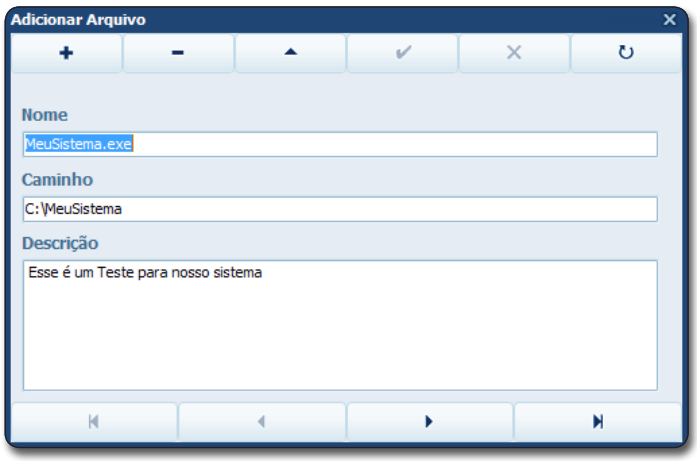


Figura 4. Formulário de cadastro e edição de arquivos

Com o formulário principal e o cadastro de arquivos criados, vamos finalizar com o envio de arquivos. Neste adicionaremos dois componentes TMyQuery, sendo o primeiro para acesso a tabela de versões (onde serão inseridos os arquivos e seus detalhes) e o segundo será responsável pela consulta a tabela de arquivos para um campo LookupComboBox, onde iremos selecionar qual o arquivo de nosso dicionário irá ser atualizado. Na query da tabela de arquivos não iremos necessitar gerar os scripts de CRUD, visto que será somente a consulta para um lookup. Para a query da tabela de versões iremos informar a instrução SQL, assim como no processo da **Figura 3**, e gerar os scripts de CRUD na tela de SQLGenerator. Após a geração do script, iremos modificar as instruções de Insert e Update na aba "Update SQLs", substituindo o parâmetro ":data_hora" por "now()". Deste modo, sempre que inserido um registro, o servidor MySQL se encarregará de informar a data automaticamente. Após concluídas essas etapas, podemos definir o layout da tela, que será simples como o da **Figura 4**. O resultado é mostrado na **Figura 5**.

Vamos definir nossa função de obtenção do MD5. Esse mesmo código será utilizado em nossa aplicação Launcher, por isso, crie uma nova unit com o nome de lib. Nela adicione em sua cláusula uses a unit IdHashMessageDigest, responsável por conter a classe de MD5 do Indy. O código finalizado é mostrado na **Listagem 2**.

Esse código é responsável por retornar o MD5 do arquivo, como descrito anteriormente. Note que a função recebe como parâmetro de entrada o nome do arquivo. Na linha 06 é criado o objeto TIdHashMessageDigest5 do Indy. Este é o principal responsável pela geração do MD5. Na linha 07 é criado um stream do arquivo

recebido por parâmetro. Na linha 09 temos o retorno que é a criação do Hash hexadecimal a partir do arquivo. As linhas 11 e 12 se encarregam de destruir e liberar os objetos da memória.

Após a definição da unit lib, iremos incluí-la na sessão uses do formulário de envio de arquivos (Alt + F11). Antes de prosseguir, adicione um componente OpenDialog ao formulário e implemente o código descrito na **Listagem 3** no evento OnClick do botão de localizar caminho (...).

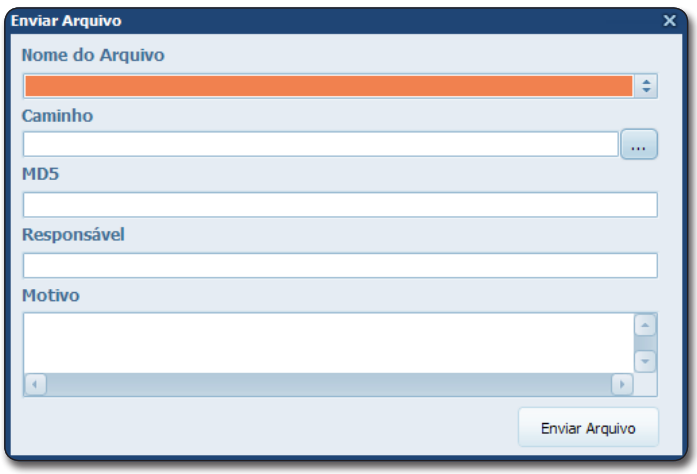


Figura 5. Formulário de envio de arquivos

Listagem 2. Função para pegar o MD5 do arquivo

```
01 function Get_MD5(const fileName : string) : string;
02 var
03  idmd5 : TIdHashMessageDigest5;
04  fs    : TFileStream;
05 begin
06  idmd5 := TIdHashMessageDigest5.Create;
07  fs    := TFileStream.Create(fileName, fmOpenRead OR fmShareDenyWrite) ;
08  try
09   result := LowerCase(idmd5.HashStreamAsHex(fs));
10  finally
11   FreeAndNil(fs);
12   FreeAndNil(idmd5);
13  end;
14 end;
```

Listagem 3. Evento OnClick do botão de localizar arquivo

```
01 procedure TFrm_EnviarArquivo.Btn_localizaArquivoClick(Sender: TObject);
02 begin
03  OpenDialog.Execute(Self.Handle);
04  Edt_Caminho.Text := OpenDialog.FileName;
05  edt_md5.EditText := Get_MD5(Edt_Caminho.Text);
06 end;
```

Observe que o código simplesmente executa o OpenDialog (linha 03). A seguir, é atribuído o caminho do arquivo ao componente Edt_Caminho (linha 04), por último, a função Get_MD5 é chamada, atribuindo o valor do hash gerado ao edt_md5 (linha 05). Note que aqui o arquivo é passado como parâmetro.

Para o botão de enviar arquivo utilizaremos um método para validação dos campos, um TypeCast no Field arquivo para TBlobField. A implementação pode ser vista na **Listagem 4**.

Listagem 4. Evento OnClick do botão de enviar arquivo

```
01 procedure TFrm_EnviarArquivo.Btn_EnviarArquivoClick(Sender: TObject);
02 begin
03  if not(Valida_Campos) then
04   Exit;
05  TBlobField(Qry_Versoes.FieldByName('arquivo')).LoadFromFile(
     Edt_Caminho.Text);
06  Qry_Versoes.Post;
07  ShowMessage('Arquivo enviado com sucesso.');
08  Close;
09 end;
```

Na implementação dessa listagem temos a checagem se os campos foram preenchidos corretamente, através da função Valida_Campos (linha 03) que retorna verdadeiro caso seja possível enviar o arquivo. Caso o retorno seja falso, o procedimento é descontinuado (linha 04), caso contrário, é realizado a importação do arquivo para o campo do tipo Blob (linha 05). Finalmente temos um post (linha 06), seguido da mensagem de êxito. A seguir a implementação da rotina Valida_Campos (**Listagem 5**).

Listagem 5. Função Valida_Campos utilizada na **Listagem 4**

```
01 function TFrm_EnviarArquivo.Valida_Campos: Boolean;
02 begin
03  Result := True;
04  if (Edt_id_arquivo.Text = '') then
05  begin
06   ShowMessage('O Campo de "Nome do Arquivo" não pode ficar em branco.
     Verifique!');
07   Result := False;
08  end
09  else if (Edt_Caminho.Text = '') then
10  begin
11   ShowMessage('O Campo de "Caminho" não pode ficar em branco.
     Verifique!');
12  Result := False;
13  end
14  else if (Edt_Responsavel.Text = '') then
15  begin
16   ShowMessage('O Campo de "Responsável" não pode ficar em branco.
     Verifique!');
17   Result := False;
18  end
19  else if (Memo_Motivo.Text = '') then
20  begin
21   ShowMessage('O Campo de "Motivo" não pode ficar em branco
     Verifique!');
22   Result := False;
23  end;
24 end;
```

Embora o código apresentado seja extenso, o processo é repetitivo. Observe que campo a campo, é checado se há conteúdo digitado nos componentes. Caso não exista, é exibida uma mensagem referente ao campo e retornado o valor False, invalidando todo o processo posterior.

Na **Listagem 6** temos os eventos OnClose e OnShow do formulário. No evento OnFormClose podemos perceber o fechamento das Queries, já no Evento OnFormShow, temos a abertura e posteriormente o Append na Qry_Versoes.

Com isso concluímos nossa aplicação de envio de arquivos. Basta definirmos uma estrutura fictícia de arquivos e enviá-los ao banco de dados para que possamos dar continuidade as próximas etapas e assim testar o Launcher posteriormente.

Listagem 6. Eventos OnFormClose e OnFormShow do formulário

```
01 procedure TFrm_EnviarArquivo.FormClose(Sender: TObject;
02   var Action: TCloseAction);
03 begin
04   Qry_Arquivo.Close;
05   Qry_Versoes.Close;
06 end;

07 procedure TFrm_EnviarArquivo.FormShow(Sender: TObject);
08 begin
09   Qry_Arquivo.Close;
10   Qry_Arquivo.Open;
11   Qry_Versoes.Close;
12   Qry_Versoes.Open;
13   Qry_Versoes.Append;
14 end;
```

Nota

Para se trabalhar com arquivos grandes no MySQL, a variável "max_allowed_packet" deve estar configurada corretamente conforme o tamanho máximo dos arquivos que se deseja armazenar. Em nosso exemplo, a variável foi definida com o valor 41943040, valor equivalente a 40 MB. Foram realizados testes enviando arquivos de até 20 MB. Lembrando que para uma otimização do funcionamento Web, a variável net_write_timeout deve estar configurada corretamente para que não ocorra erros de perda de conexão enquanto o arquivo é gravado.

Criando o Launcher

Com o sistema de envio de arquivos finalizado e com os arquivos devidamente carregados na base de dados, vamos dar início ao aplicativo Launcher. Para isso vamos criar uma classe simples que vai acessar o banco de dados, consultar a tabela de versões e percorrê-la verificando os arquivos nos devidos diretórios. Essa classe será composta de um componente de conexão com a base de dados e um método de validação que receberá dois parâmetros de entra, sendo eles: um Caption que informará ao usuário o status da atualização. O segundo será uma barra de progresso (ProgressBar) que exibirá o status da atualização. Esta classe será definida na unit lib sua definição pode ser vista no código da **Listagem 7**.

Após a definição da classe, iremos dar início a parte de implementação. A **Listagem 8** demonstra o código utilizado para nosso construtor definido anteriormente na linha 11 da **Listagem 7**.

No código podemos notar que a conexão é de fato criada (linha 03) e posteriormente configurada (linhas 04 a 08), por fim, a mesma é aberta (linha 09).

Continuando a implementação da classe TValida_Arquivo, a **Listagem 9** exibe o principal e mais longo método da classe de mesmo nome. Embora extenso, o mesmo é simples, veremos a seguir o que é realizado no processo.

Listagem 7. Código da unit lib

```
01 unit lib;

02 interface

03 uses

04  Windows, Forms, Messages, SysUtils, Variants, Classes, MyAccess,
05  IdHashMessageDigest,Vcl.StdCtrls, Vcl.ComCtrls, Data.DB;

06 type
07  TValida_Arquivo = class
08  private
09    vConnection : TMyConnection;
10  public
11    constructor Create;
12    destructor  Destroy;
13    function valida_arquivos(pCaption : TLabel;
      ProgressBar: TProgressBar) : Boolean;
14  end;

15  function Get_MD5(const pFileName : String) : string;

16  implementation
```

Listagem 8. Método construtor

```
01 constructor TValida_Arquivo.Create;
02 begin
03    vConnection := TMyConnection.Create(Nil);
04    vConnection.Database := 'dic_estrutura';
05    vConnection.Port     := 3306         ;
06    vConnection.Server   := 'Meu_Servidor' ;
07    vConnection.Username := 'Meu_Usuario'  ;
08    vConnection.Password := 'Minha_Senha'  ;
09    vConnection.Connect;
10 end;
```

Podemos notar no código que na linha 07 é criada a query responsável por consultar a tabela de arquivos. Entre as linhas 12 a 21, temos a inclusão das instruções SQL, na linha 22 percebe-se a abertura desta Query. Temos posteriormente a demarcação do início da barra de progresso (linha 23), bem como a posição máxima, que atribuída de acordo com o número de registros obtidos na consulta (linha 24). A seguir, entre as linhas 26 a 39 temos um laço. Perceba que a posição do ProgressBar é incrementada de acordo com o número do registro atual do DataSet (linha 28). Na linha 30, o Caption passado por parâmetro de entrada, é atualizado com o nome do arquivo a ser verificado. Posteriormente, na linha 32, a variável vPath recebe a concatenação do caminho com o nome. Na linha 33 o campo md5 é validado em comparativo à variável vPath (utilizando a função Get_MD5, anteriormente criada). Caso ambos sejam diferentes, é realizada a atualização do blob por meio da importação do arquivo (vPath - linha 37). Por último, a atualização é concluída e a consulta finalizada (linhas 43 e 44).

Realizadas as codificações na classe, daremos início ao formulário principal da aplicação. Crie um novo formulário e salve-o conforme desejar, nele adicione um componente Timer. Adicione também ao uses do formulário a nossa unit Lib definida anteriormente.

Listagem 9. Função Valida_Arquivos

```
01 function TValida_Arquivo.valida_arquivos(pCaption : TLabel;
     ProgressBar: ProgressBar) : Boolean;
02 var
03   vConsulta : TMyQuery;
04   vPath     : String;
05 begin
06   Result := False;
07   vConsulta := TMyQuery.Create(vConnetion);
08   try
09     vConsulta.Connection := vConnetion;
10     vConsulta.Close;
11     vConsulta.SQL.Clear;
12     vConsulta.SQL.Add(' Select              ');
13     vConsulta.SQL.Add(' nome   ,            ');
14     vConsulta.SQL.Add(' caminho,            ');
15     vConsulta.SQL.Add(' md5    ,            ');
16     vConsulta.SQL.Add(' arquivo             ');
17     vConsulta.SQL.Add(' From arquivos ');
18     vConsulta.SQL.Add(' Left Join versoes on
       (versoes.id_arquivo =
       arquivos.id_arquivo)                    ');
19     vConsulta.SQL.Add('Where versoes.id_versao = (Select
       max(Max_Ver.id_versao)                  ');
20     vConsulta.SQL.Add('From versoes Max_Ver ')
21     vConsulta.SQL.Add('Where Max_Ver.id_arquivo = arquivos.id_arquivo)');
22     vConsulta.Open;
23     ProgressBar.Position := 0;
24     ProgressBar.Max    := vConsulta.RecordCount;
25   vConsulta.First;
26   while not(vConsulta.Eof) do
27   begin
28     ProgressBar.Position := vConsulta.RecNo;
29     Application.ProcessMessages;
30     pCaption.Caption := 'Verificando o arquivo ' +
       vConsulta.FieldByName('nome').AsString;
31     Application.ProcessMessages;
32     vPath := vConsulta.FieldByName('caminho').AsString + '\' +
       Consulta.FieldByName('nome').AsString;
33     if (vConsulta.FieldByName('md5').AsString <> Get_MD5(vPath)) then
34     begin
35       pCaption.Caption := 'Atualizando o arquivo ' +
         Consulta.FieldByName('nome').AsString ;
36       Application.ProcessMessages;
37       TBlobField(vConsulta.FieldByName('arquivo')).SaveToFile(vPath);
38     end;
39     vConsulta.Next;
40   end;
41   Result := True;
42 finally
43   pCaption.Caption := 'Atualização concluida';
44   FreeAndNil(vConsulta);
45 end;
46 end;
```

A maioria das alterações deste formulário são visuais, sendo assim, o modelo proposto é mostrado na **Figura 6**. Iremos definir no formulário principal uma procedure chamada Valida_Arquivos, que será chamada no evento OnTimer do componente de mesmo nome. Sua implementação encontra-se definida na **Listagem 10.**

Figura 6. Formulario de envio de arquivos.

Listagem 10. Código da procedure Valida_Arquivos

```
01 procedure TFrm_Main.Valida_Arquivos;
02 var Valida_Arquivos : TValida_Arquivo;
03 begin
04  Valida_Arquivos := TValida_Arquivo.Create;
05  try
06   if (Valida_Arquivos.valida_arquivos(Lbl_Progresso,ProgressBar)) then
07   begin
08    ShellExecute(Application.Handle,'open',
      'C:\MeuSistema\Minha_aplicacao.exe',
      'aberto_via_launcher'',SW_SHOWNORMAL);
09    Application.Terminate;
10   end
11   else
12   begin
13    if (MessageDlg('Ocorreu algum problema com sua atualização.'
14      + #13#10 +'Deseja tentar uma nova atualização?',
      mtConfirmation,[mbYes, mbNo],6) = 6) then
15     Self.Valida_Arquivos
     else
16     Application.Terminate;
17   end;
18  finally
19   FreeAndNil(Valida_Arquivos);
20  end;
22 end;
```

Perceba pelo código que na linha 04 foi criada a classe TValida_Arquivo, a seguir, na linha 06 é chamado o método Valida_Arquivos da classe de mesmo nome (passando como parâmetros o Label de progresso e o componente ProgressBar). Na linha é utilizado o ShellExecute com a instrução de executar o sistema (caso o retorno da função seja True), encerrando a aplicação em seguida. Caso o resultado da função seja oposto, é exibida uma mensagem de erro com confirmação, propondo uma nova atualização (linhas 13 e 14), realizando assim uma nova execução da rotina, ou finalizando a aplicação. Na linha 19 a nossa classe é destruída e liberada da memória.

Abrindo a aplicação somente pelo Launcher

Para abrir a aplicação somente se um determinado parâmetro for passado é razoavelmente simples. Basta que verifiquemos se uma determinada posição do vetor de parâmetros é o valor desejado. Para isto, utilizarmos a função ParamStr. Na **Listagem 11** temos um exemplo de seu uso que permite utilizar o Handle da janela (identificador de acordo com o nome do aplicativo).

Listagem 11. Exemplo de verificação do parâmetro via Handle

```
01 {Verifica se o parametro foi passado}
02 if not(ParamStr(1) = 'aberto_via_launcher') then
03 begin
04  ShowMessage('Esta aplicação deve ser aberta pelo Launcher!');
05  Application.Terminate;
06  Exit;
07 end;
08 ShowMessage('Inciando a aplicação');
```

Note que na linha 02 é verificado se existe a string de acordo com ParamStr(1) (responsável por validar com o nome do aplicativo). Caso o resultado da validação seja verdadeiro (diferente), uma mensagem é exibida e a aplicação é encerrada (linhas 04 a 06).

Sugestões

Outra forma possível de armazenar os arquivos seria uma solução baseada em servidor FTP, utilizando uma suíte como Indy por exemplo.

Uma sugestão a ser implementada no Launcher é um informativo das alterações que ocorreram, para que os usuários tenham conhecimento das atualizações. Além disso, pode-se implementar um painel como um número "x" de imagens ou propagandas de outros produtos entre outros.

Autor

Gutierry Antonio
Gutierrydsn@hotmail.com
Software Engineer \ MySQL DBA
Cursando o ensino superior em Sistema da informação e Analise de Sistema na Universidade Estácio de Sá. Experiência em desenvolvimento com Visual Fox Pro, Delphi, Ruby on Rails, Action Script, Java, C# e C++, bem como vários bancos de dados, como Firebird, MySQL, SQL Server e Oracle.

Links:

MyDac – Conheça um pouco sobre o MyDac
http://www.devart.com/mydac/

Você gostou deste artigo?

Dê seu voto em ***www.devmedia.com.br/clubedelphi/feedback***

Ajude-nos a manter a qualidade da revista!

Rapid SQL XE5: Conheça seu Banco de Dados

Informações essenciais sempre à mão

O Rapid SQL é uma ferramenta da Embarcadero voltada ao trabalho com banco de dados. De forma mais abrangente, assim como o Delphi, pode ser definido como um ambiente de desenvolvimento integrado, que remete ao termo em inglês Integrated Development Environment (IDE). Logo, isso implica dizer que num mesmo ambiente, o desenvolvedor terá à sua disposição uma série de recursos distintos, tal qual um agregado de ferramentas paralelas. Como exemplo, dentre as aptidões providas de forma pontual pelo Rapid SQL estão o gerenciamento dos objetos de banco de dados, a criação de scripts SQL, engenharia reversa, controle de versão, etc.

Em seu estágio atual, a ferramenta conta com uma nova versão recém-lançada que, assim como o Delphi, apresenta-se sob o release denominado XE5. Apesar de apresentar novidades pontuais, grande parte deste cenário evolutivo do Rapid SQL se dá sobre o amadurecimento da ferramenta como produto, principalmente por meio de melhorias de seus principais recursos já existentes. Em vista disso, o artigo extrapola o cenário das novidades, apresentando ao leitor um panorama geral sobre a ferramenta.

SGBDs tradicionalmente suportados

Em termos de SGBD, tanto Interbase quanto Firebird, podem ser plenamente operados pelo Rapid SQL. No caso do primeiro, sua versão mais recente nominalmente suportada pela ferramenta é a XE3 (Interbase XE3), enquanto que para o segundo é a 2.0 (Firebird 2.0). Contudo, sem deixar de mencionar, a ferramenta prevê ainda o suporte a diversos outros SGBDs do mercado, tais como Oracle, DB2 e MS SQL Server. A seguir são então listadas as versões mais recentes suportadas de alguns deles:

- IBM DB2 z/OS 9 e LUW 9.7;
- Microsoft SQL Server 2012;
- MySQL 5;
- Oracle 11g;
- Sybase ASE 15.5 e IQ 15.4;

Fique por Dentro

O cotidiano de grande parte dos desenvolvedores Delphi inclui, além do próprio software de desenvolvimento, algumas outras ferramentas tidas como complementares, tais quais as relacionadas à manipulação de bancos de dados. Isso se justifica pelo fato dessas tratativas serem, em muitos casos, direcionadas também ao programador. Sendo assim, a escolha por uma ferramenta capaz de simplificar essas tarefas de forma eficaz, acaba por se tornar um item essencial nos dias atuais, onde a questão produtividade é algo vital a qualquer tipo de negócio. O presente artigo traz como foco então a ferramenta Rapid SQL, que visa atender a todos estes requisitos, provendo recursos suficientes para se operar diversos tipos de bancos de dados, tais quais InterBase e Firebird (apenas para citar dois), que podem ser considerados dois dos principais SGBDs presentes no contexto das aplicações Delphi.

Áreas de abrangência

O Rapid SQL, como ferramenta, abrange várias áreas distintas relacionadas ao desenvolvimento de banco de dados. No caso específico de Interbase e Firebird, isso resulta em diversos pontos de atuação, essenciais para a manipulação de bancos de dados desse tipo.

Gerenciamento de fontes de dados

Uma vez instalado, o Rapid SQL permite o registro de uma fonte de dados (data source) à sua interface e, por conseguinte, sua manipulação e gerenciamento. Dessa forma, tais quais outras ferramentas do gênero, há uma centralização de acesso a diversas fontes distintas, a partir de um mesmo local de uso. Esta segmentação acaba por formalizar um catálogo de fontes, o que inclui tanto armazenamentos locais quanto remotos.

Gerenciamento de objetos de banco de dados

O gerenciamento a fontes de dados provido pela ferramenta está naturalmente associado à gestão dos objetos que compõem a estrutura do banco de dados. Para isso, o Rapid SQL provê o suporte a uma ampla gama de tipos de objetos diferentes em reflexo aos seus diversos tipos de SGBDs suportados. A partir disso, através de sua interface, o desenvolvedor torna-se apto a

realizar diversas ações sobre tais objetos, tal como criar um novo ou mesmo editar/excluir um existente.

Trabalho com SQL

A SQL é tida como a linguagem predominante e padrão para pesquisas declarativas em bancos de dados relacionais, sendo assim, o Rapid SQL provê um ambiente completo com recursos dedicados à produção e manipulação de consultas e scripts deste tipo. Auxiliares de codificação, assistentes de semântica e validadores de sintaxe são alguns dos exemplos encontrados. Por fim, a ferramenta contempla ainda facilidades no que diz respeito à carga e execução de arquivos relacionados.

Trabalho com dados

Ao longo de sua trajetória, o Rapid SQL sempre primou por apresentar constantes melhorias em suas principais funcionalidades. Dentre elas, um destaque maior pode ser dado ao seu editor de dados, que acaba por caracterizar uma das figuras de maior excelência na interface de suas mais recentes versões, visto a facilidade de uso provida e recursos relacionados. Logo, o editor permite a alteração dos dados da tabela de forma visual e em tempo real, suportando os mais variados tipos de dados, incluindo opções Unicode. Como adicional, possibilita também a exportação e importação das informações em diversos formatos. Neste ponto, o uso de planilhas pode ser considerado uma das abordagens mais tradicionais.

Controle de versão

Pelo fato da essência da manipulação de banco de dados envolver processos como alteração em objetos e produção de scripts SQL, surge a necessidade natural da gestão de tais requisitos. Visando atender demandas como esta, o Rapid SQL conta com funcionalidades que abrangem o gerenciamento de projetos de banco de dados, o que inclui um eficiente controle de versão, mantido em um repositório central, provendo toda a rastreabilidade de alterações em objetos e código-fonte. Conceitualmente, no contexto da ferramenta, cada banco de dados gerenciado dá origem a um projeto distinto.

Toda essa característica se aproxima muito ao cenário Delphi, cujo IDE também contempla um controle de versão próprio para seus projetos, provido por meio de uma ferramenta externa, o Subversion (ver **BOX 1**).

BOX 1. Subversion

De um modo geral, um sistema de controle de versão propicia um controle eficiente sobre o histórico de mudanças de determinado arquivo, armazenando tais informações em um repositório centralizado e seguro. Conceitualmente, o Subversion pode ser definido como um sistema de controle de versão de código aberto, originado de um projeto colaborativo homônimo fundado na década de 2000 pela CollabNet e que atualmente é mantido como um projeto da Apache Software Foundation. Seu uso envolve o versionamento de arquivos dos mais variados tipos, com destaque aos de conteúdo em texto, o que inclui arquivos de código-fonte. Sua integração com o Delphi se deu ainda na versão XE do IDE, sendo um dos grandes atrativos deste release.

Facilitadores para atividades rotineiras

A interface do Rapid SQL conta também com uma gama de pequenos utilitários que acabam por facilitar algumas das atividades simples tidas como "de rotina", que permeiam o cotidiano do desenvolvedor. Como exemplo, é possível citar a presença de ferramentas como: buscador de palavras, buscador de arquivos, comparador visual de diferenças entre arquivos ou objetos de banco de dados e acesso direto ao agendador de tarefas (task scheduler) do Windows.

Análise e depuração de código

O Rapid SQL conta ainda com ferramentas internas para análise e depuração de código, que estão relacionadas fundamentalmente a dois tipos de objetos: Stored Procedures e Functions. A primeira, intuitivamente identificada como Code Analyst, busca identificar no corpo dos objetos destes tipos, eventuais linhas de código que estão consumindo um tempo maior que o desejado. Já a segunda está disponível apenas para um grupo seleto de SGBDs (especificamente IBM DB2, Oracle, Sybase e MS SQL Server), permitindo se realizar testes mais detalhados sobre procedimentos e funções.

Conectividade

Conforme mencionado, o Rapid SQL provê suporte e, consequentemente, conectividade a uma diversa gama de SGBDs diferentes. Para tal, a ferramenta habilita duas formas distintas para que isso aconteça. A primeira é a mais natural e tida como "padrão", uma vez que lida com o aspecto nativo provido pelo software. Neste cenário, o único requisito relevante se dá pela presença do cliente do SGBD ou da combinação cliente/driver, devidamente instalados e configurados no ambiente relacionado. Para cada SGBD a ferramenta pode exigir uma combinação de requisitos diferentes. Como exemplo, tomando como base o Firebird, que é um dos principais SBGDs relacionados às aplicações Delphi, seu uso no Rapid SQL envolve apenas a instalação de seu driver ODBC (Firebird ODBC driver) oficial. Já para o Interbase, que é o banco de dados da própria Embarcadero, a necessidade se dá pela presença do driver ODBC específico (DataDirect ODBC driver for Interbase), bem como do Interbase Client. Sem deixar de mencionar, a seguir são listados os requisitos exigidos pelos demais SGBDs suportados:

- Oracle: Oracle Sql*Net Client;
- Sybase ASE: Sybase Open Client;
- Sybase IQ: SQL Anywhere ODBD driver / Sybase IQ 32-bit ODBC driver;
- Microsoft SQL Server: Microsoft SQL Server Client Library;
- MySQL: MySQL Connector/ODBC Driver;
- IBM DB2 for z/OS: DB2 UDB Client for Windows;
- IBM DB2 for Linux, Unix, Windows: IBM DB2 LUW Client for Windows.

A outra forma de conectividade provida pela ferramenta envolve o uso de drivers de terceiros, mais especificamente drivers do tipo

JDBC (ver **BOX 2**). Neste caso, novamente tendo o Firebird como exemplo, o requisito necessário para seu uso com o Rapid SQL se dá pelo uso do driver Jaybird JDBC. Já para Interbase, o elemento associado é o Interbase JDBC Driver. Seguindo por esta linha, a seguir são listados outros exemplos relacionados aos demais SGBDs suportados:

- Oracle: Oracle JDBC Thin Driver;
- Sybase ASE: Sybase jConnect JDBC Driver;
- Sybase IQ: Sybase jConnect JDBC Driver for Sybase IQ;
- Microsoft SQL Server: jTDS Type 4 JDBC Driver for Microsoft SQL Server;
- MySQL: MySQL Connector/J JDBC Driver;
- IBM DB2 for z/OS: IBM Data Server Driver for JDBC;
- IBM DB2 for Linux, Unix, Windows: IBM Data Server Driver for JDBC.

BOX 2. JDBC

JDBC é o acrônimo para o termo Java DataBase Connectivity e que de forma breve pode ser definido como sendo um conjunto de classes e interfaces escritas na linguagem Java, que lidam com a conectividade a diversos bancos de dados relacionais. Em vista disso, cada SGBD apresenta um driver JDBC próprio e específico.

Profissionais beneficiados

No contexto do desenvolvimento de software, o uso do Rapid SQL proporciona benefícios a diversos tipos de profissionais. O primeiro deles, tal qual é o enfoque deste artigo, se reflete nos próprios desenvolvedores (Developers) Delphi, que podem então desfrutar de todas as facilidades providas pela ferramenta, agilizando diversos pontos de seu cotidiano de trabalho. Um exemplo primário se dá pela rápida produção de códigos e scripts SQL, que são dois dos elementos essenciais ao trabalho do desenvolvedor com banco de dados. Em complemento a este cenário, estão outros dois tipos de profissionais: os de banco de dados e gerentes de projeto.

Diante de todas as características da ferramenta que, por si só, já acabam por justificar seu uso, sua interface gráfica intuitiva e sua pesquisa automática por dependências durante a alteração de um objeto, podem ser pontos atrativos a um Administrador de Banco de Dados. Ao mesmo tempo em que seu Controle de Versão colabora num aspecto de gestão fundamental a projetos de banco de dados.

Segmentações do produto

O Rapid SQL, como produto, é segmentado perante dois aspectos, que impactam diretamente na forma coma a ferramenta é disponibilizada comercialmente ao grande público. Sendo assim, o primeiro aspecto diz respeito à edição do produto que, no presente cenário, se divide em Standard Edition e Professional Edition. Conforme sua nomenclatura sugere, Standard é a edição de entrada do produto, fornecendo uma linha de recursos básicos ao utilizador, tais como a gestão de fontes de dados, gestão de objetos e trabalho com SQL (edição e execução). Em complemento, a edição Professional provê todos os mesmos recursos da Standard, com o adicional de utilitários pontuais como SQL Debugger, SQL Profiler e Code Analyst.

O segundo aspecto de segmentação do Rapid SQL diz respeito à plataforma de banco de dados que o produto irá atender. Dessa forma, neste cenário, novamente a ferramenta apresenta duas opções, caracterizadas por um direcionamento mais específico ou mais amplo, em termos de plataforma atendida. Cada opção desta é tida comercialmente como uma "versão" diferente. De forma básica, cada SGBD suportado dá origem a uma versão "específica", tal como Rapid SQL for Oracle, Rapid SQL for SQL Server e Rapid SQL for DB2. Adicionalmente, de uma forma mais ampla, a versão XE Pro (Rapid SQL XE) engloba o trabalho com todos os SGBDs suportados a partir de uma única interface.

Requisitos exigidos

Por ser uma ferramenta considerada utilitária, o Rapid SQL exige muito pouco, em termos de requisitos de hardware, para ser instalado. Pontualmente, 1GB de memória e de espaço em disco podem ser estabelecidos como os itens básicos. Já sob a perspectiva do sistema operacional, o aplicativo, por ser um software para Windows, garante sua plena funcionalidade a partir da versão XP deste. Em ambientes 64 bit, a ferramenta, até sua versão XE4, era executada de forma emulada, uma vez que possuía características essencialmente 32 bit. Neste cenário, de uma forma geral, o único detalhe ficava por conta do Rapid SQL somente prover suporte à versão 32 bit do software cliente do SGBD manipulado. Todavia, todo este cenário sofre uma evolução nesta versão XE5, conforme será visto mais adiante.

Instalação

Como todo software comercial, o processo de instalação do Rapid SQL dá-se início, primeiramente, na definição do tipo de licença a ser utilizada. Para este caso, assim como ocorre em toda a gama de produtos da Embarcadero, as opções disponibilizadas ao grande público são caracterizadas por alternativas com e sem custos. Logo, a opção paga é tida como "completa" enquanto que a gratuita é voltada a critérios de avaliação, sendo denominada "Trial". Tradicionalmente, versões deste tipo proveem aos usuários um período de 30 dias corridos para testes, a partir da instalação do software no sistema. No caso do Rapid SQL, este tempo é ajustado para 14 dias.

Naturalmente, para a escrita deste artigo, será levada em conta a versão Trial do produto, que está acessível a toda e qualquer pessoa. O processo operacional de instalação como um todo segue o padrão de Wizard (Next-Next-Finish) do Windows, não sendo necessário nenhum comentário adicional. Uma vez concluída tal etapa, a primeira execução do Rapid SQL XE5 contempla uma janela de diálogo de opções, conforme mostra a **Figura 1**, as quais irão contribuir para a ideal utilização da ferramenta por parte do usuário.

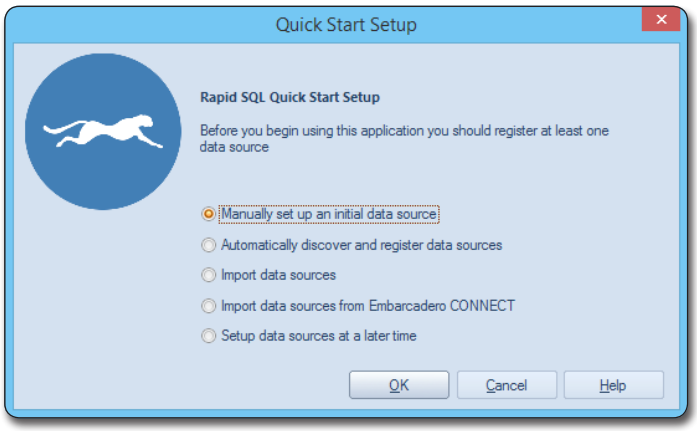


Figura 1. Rapid SQL XE5 – Diálogo de opções iniciais

Ambiente

Desde suas versões mais iniciais até sua mais recente, o Rapid SQL apresenta um mesmo estereótipo de ambiente e interface. Logo, nesta sua versão XE5, a base padrão é mantida, com o aspecto do IDE estabelecido em cinco pilares – Database Explorer (1), Workspace (2), Output Window (3), Toolbar (4) e Menus (5) – conforme pode ser visto na **Figura 2**. Toda esta estilização inicial é, obviamente, complementada por elementos extras, acionados pelas diversas opções providas pela ferramenta. A fim de se firmar conceitos, a seguir são elencados alguns dos itens citados bem como outros relacionados.

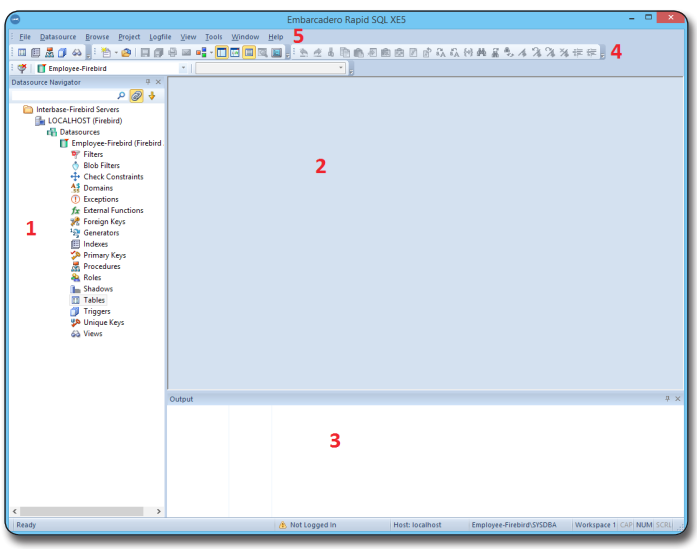


Figura 2. Rapid SQL XE5 – Ambiente

Database Explorer

Tido como um dos, senão o principal, elemento do IDE, o Database Explorer nada mais é do que a estrutura (uma simples janela) onde são organizadas, em forma de árvore, todas as informações relevantes geridas pela ferramenta, fundamentalmente relacionadas aos servidores de banco de dados, e disponíveis ao usuário. Sob o aspecto prático, o Database Explorer provê uma forma ágil de acesso aos scripts e objetos de banco de dados. Por padrão, assim como ocorre no ambiente do Delphi, elementos de interface como esse são manipuláveis e acopláveis (dockable), de forma que o utilizador pode estabelecer sua posição por diferentes pontos do IDE.

Workspace

Quase que em sua totalidade, as ferramentas relacionadas ao âmbito do desenvolvimento de software trazem em sua arquitetura um elemento central tido como o "palco principal" de sua interface de trabalho. Neste local é onde serão então realizadas grande parte das atividades rotineiras e essenciais desenvolvidas no aplicativo. Numa exemplificação inicial, citando o próprio Delphi, seu IDE apresenta o Form Designer/Code Editor como sendo sua área principal de trabalho, ou seja, a porta de entrada para o desenvolvimento das atividades na ferramenta que, neste caso, se trata da construção do layout das telas e da codificação propriamente dita.

Por sua vez, o palco principal do Rapid SQL é denominado Workspace, que numa tradução livre para o português seria algo como "espaço de trabalho". Fisicamente, ele acaba por englobar toda a área de atividade da ferramenta, representado por toda sua área da direita, tal como mostrado na figura anterior. Adicionalmente, em suas versões mais recentes, incluindo a atual XE5, o Rapid SQL provê o trabalho com múltiplos Workspaces (até três, por padrão). Essa abertura possibilita então uma ideal divisão entre as diversas tarefas realizadas na ferramenta, através da consequente distribuição destas a cada Workspace. Como exemplo, numa situação real, scripts de diferentes segmentos podem ser direcionados a "espaços de trabalho" distintos, proporcionando ganhos que vão desde a organização visual obtida à contribuição num melhor entendimento das atividades.

Describe Window

Conforme seu próprio nome propõe, a janela Describe Window tem como finalidade descrever informações sobre determinados tipos de objetos de banco de dados que estão sendo mostrados na interface do Rapid SQL. Logo, tais tipos envolvem exclusivamente quatro elementos básicos que permeiam o contexto das bases de dados dos mais variados SGBDs: tabelas (tables), visões (views), procedimentos (stored procedures) e funções (functions). Na prática, as informações de tais objetos são dispostas na interface do Describe Window na forma de tabela, conforme pode ser visto na **Figura 3**. Sem deixar de mencionar, para cada tipo de objeto, a janela Describe exibe um conjunto distinto de informações:

- Tables e Views: para cada objeto do tipo Table ou View, é exibida sua lista de campos, cada qual com seu respectivo tipo de dado, obrigatoriedade (se aceita valores null) e valor padrão (default value);
- Procedures e Functions: para cada objeto do tipo Procedure ou Function, é exibida sua lista de parâmetros, cada qual com seu respectivo tipo de dado e modo (se de entrada ou saída).

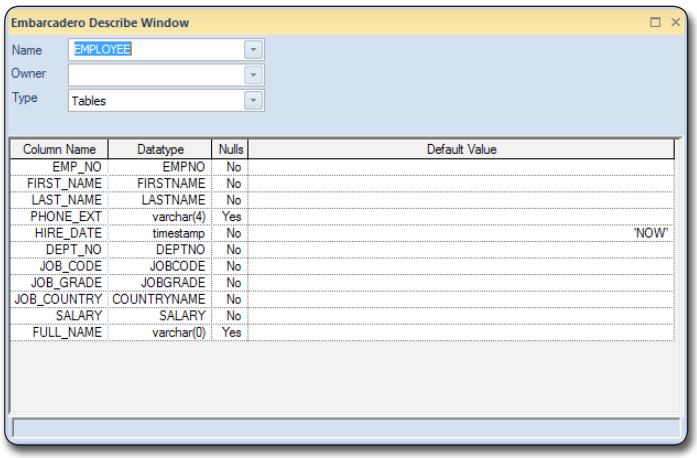


Figura 3. Rapid SQL XE5 – Describe Window

Output Window

O ambiente padrão do Rapid SQL conta ainda com um elemento denominado Output Window (**Figura 4**), essencial às intenções da ferramenta. Isto porque, esta janela é voltada à captura de todas as mensagens resultantes das diversas operações executadas no IDE, tais como uma informação retornada do servidor de banco de dados ou mesmo uma simples estabilização de conexão. Novamente traçando um paralelo com o Delphi, esta estrutura pode ser comparada à sua janela de mensagens (Messages) que, entre outras coisas, expõe as mensagens do compilador e de eventuais erros no código.

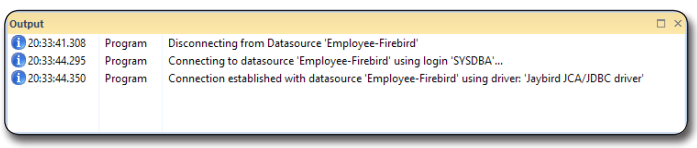


Figura 4. Rapid SQL XE5 – Output Window

Utilitários internos

Apesar do Rapid SQL ser considerado uma ferramenta para projetos de bancos de dados, internamente, seu próprio ambiente contempla uma série de pequenos recursos auxiliares que, de forma conceitual, também são tidos como utilitários internos. Conforme já citado anteriormente, a área de abrangência de tais recursos envolvem operações dos mais variados tipos, das extremamente simples às mais complexas. Fisicamente, tais utilitários são disponibilizados através do menu Tools do IDE (**Figura 5**). Nominalmente, a seguir são expostos alguns deles:

Find in Files

Disponível no menu Tools, a opção Find in Files, assim como seu nome sugere, dá origem a uma ferramenta (**Figura 6**) para encontrar palavras ou expressões em arquivos diversos. Na caixa de diálogo que se abre, toda sua funcionalidade é essencialmente baseada no provimento de três informações, refletidas nos seguintes campos:

- Find what: campo a ser usado para se especificar a palavra, caractere, expressão ou frase que se deseja encontrar;
- In files/file: especifica o tipo do(s) arquivo(s) que farão parte do processo de varredura, tais como as tradicionais extensões ".sql" e ".txt";
- In folder: indica o diretório onde estarão localizados os arquivos envolvidos.

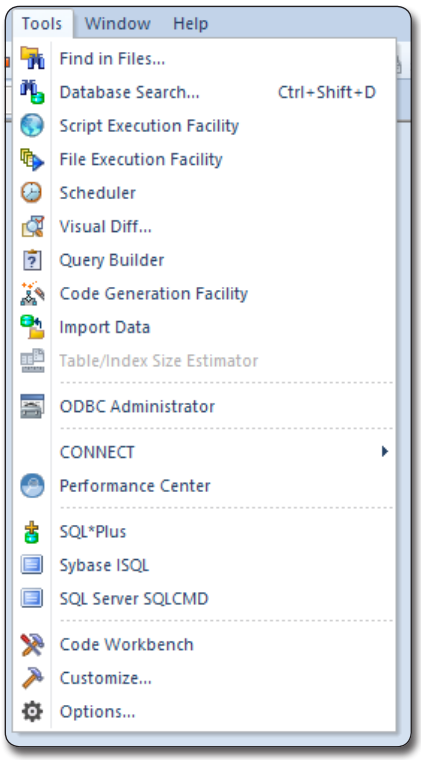


Figura 5. Rapid SQL XE5 – Menu Tools

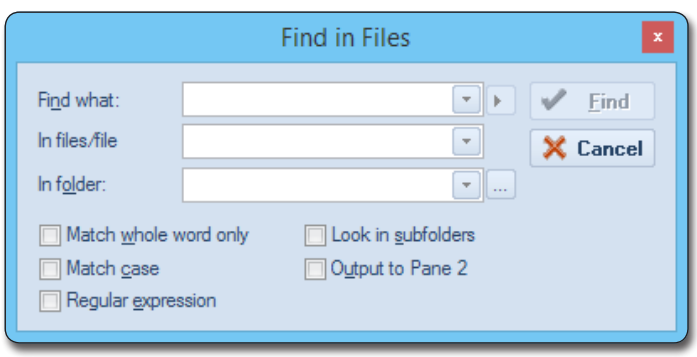


Figura 6. Find in Files

Database Search

De forma semelhante à funcionalidade provida pelo recurso Find in Files que, basicamente, contempla a pesquisa de um texto no conteúdo de arquivos externos, o utilitário Database Search tem seu processo de busca voltado aos bancos de dados registrados na ferramenta. Esta opção, ao ser acionada no menu Tools | Database Search, dá origem a um rápido assistente (Wizard) de

duas etapas. Na primeira, o usuário deverá selecionar as fontes de dados de interesse, dentre as registradas no IDE do Rapid SQL, que sofrerão a ação de pesquisa. Em complemento, na segunda e última etapa, o assistente guia o usuário pela definição dos critérios de busca, tal qual o termo a ser procurado, bem como os tipos de objetos participantes do processo (Ex: functions, procedures, tables, etc.), que terão seu código DDL vasculhado. A seguir, a **Figura 7** ilustra o resultado de uma busca pela palavra "EMPLOYEE" pelos objetos do banco Employee do Firebird. Conforme pode ser visto, para este caso, foram encontradas ocorrências nas instruções DDL de dois objetos Table (EMPLOYEE e EMPLOYEE_PROJECT) e quatro objetos Procedure (ADD_EMP_PROJ, DELETE_EMPLOYEE, GET_EMP_PROJ e ORG_CHART).

Visual Difference

Visual Difference é o nome do utilitário provido pelo Rapid SQL que possibilita a comparação visual entre dois arquivos ou objetos de banco de dados. Logo, sua utilidade se reflete na análise e posterior sincronização dos elementos envolvidos. Na prática, o contexto fundamental deste recurso se resume a uma caixa de diálogo contendo dois painéis (**Figura 8**), onde os conteúdos de cada arquivo ou objeto são então postos lado a lado, tendo suas eventuais diferenças ressaltadas. Conceitualmente, o painel da esquerda faz referência ao arquivo/objeto "fonte" (source), enquanto que o da direita se refere ao arquivo/objeto de "destino" (target). Outro detalhe desta caixa de diálogo fica por conta da presença de uma barra de ferramentas (toolbar) própria, contendo opções relacionadas, tais como:

- Print: provê a impressão do conteúdo de cada painel da caixa de diálogo do Visual Difference;
- Navigate Differences: possibilita a navegação por entre as diferenças detectadas, utilizando-se os botões de seta para cima (anterior) e para baixo (próxima);
- Search: provê a pesquisa de um determinado texto por entre o conteúdo dos objetos/arquivos comparados.

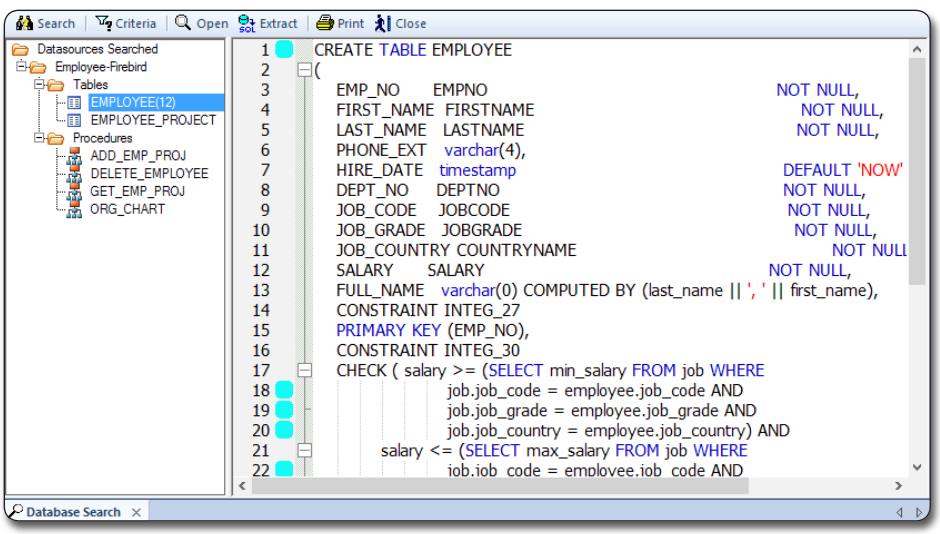


Figura 7. Database Search – Tela de resultados

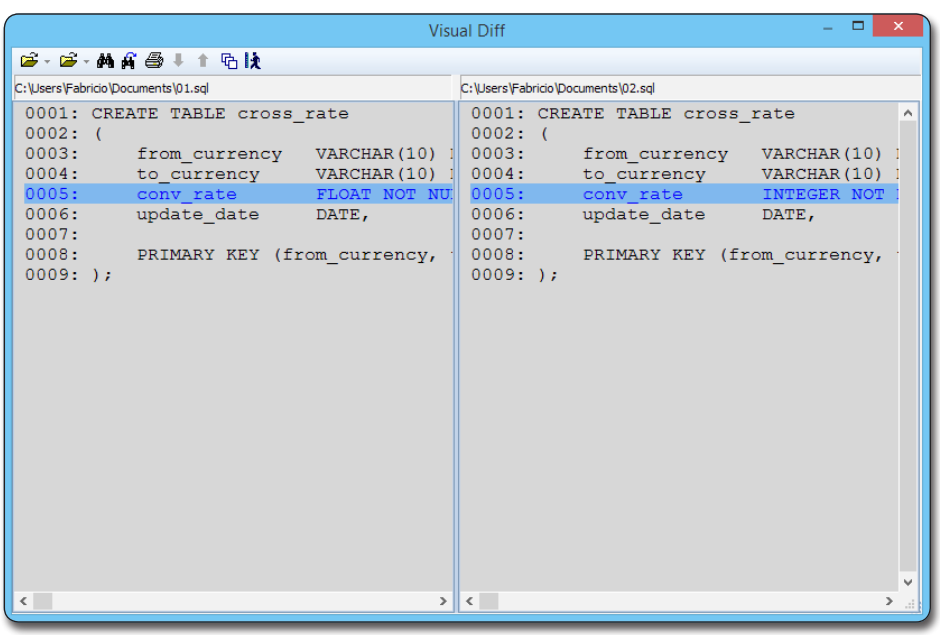


Figura 8. Visual Difference

Query Builder

O utilitário Query Builder (**Figura 9**), através de sua interface, provê uma forma totalmente visual e intuitiva de se construir instruções SQL, atendendo a todos os SGBDs suportados pelo Rapid SQL. Aqui tais instruções são essencialmente voltadas à manipulação de dados, mais precisamente do vocabulário DML (Data Manipulation Language – Linguagem de Manipulação de Dados) da linguagem SQL. Logo, estas podem ser segmentadas em cinco tipos distintos, todas relacionadas ao padrão convencional, incluindo as instruções básicas SELECT, INSERT, UPDATE e DELETE, além da complementar CREATE VIEW. Logo, a construção de cada uma delas se dá de forma totalmente sincronizada com a manipulação visual que está ocorrendo, podendo ser considerado então um utilitário "online". De acordo com seu paradigma, o Query Builder prevê ainda a execução direta, através de sua própria interface, do comando SELECT criado.

Devido as suas características, um dos principais propósitos de uma ferramenta como o Query Builder é abstrair de seu utilizador o conhecimento da linguagem padrão de consultas em bancos de dados, a SQL. Num exemplo claro e inicial, através de sua utilização, qualquer usuário é capaz de construir uma consulta a dados envolvendo múltiplas tabelas relacionadas, sem um conhecimento prévio específico determinante (ex: uso de JOINS).

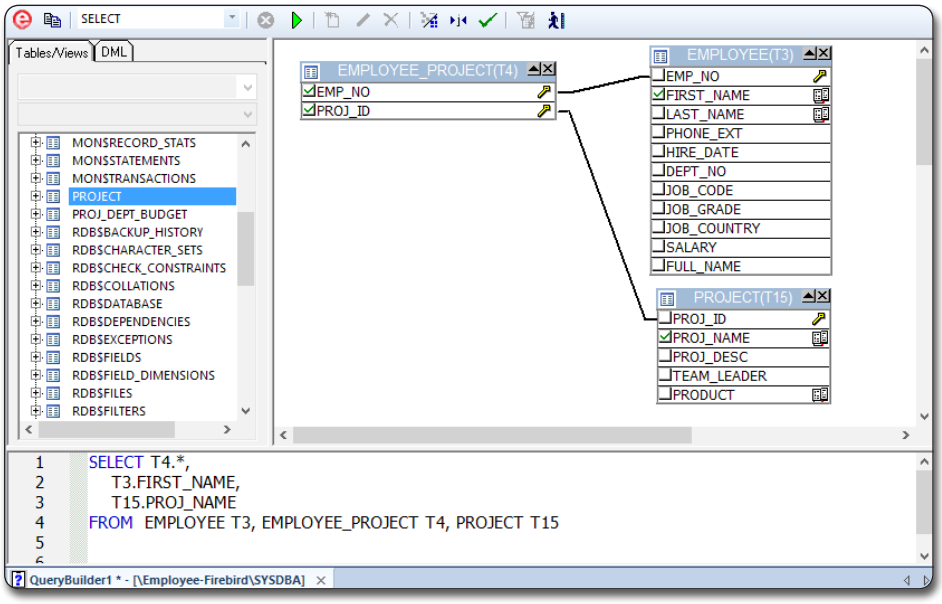


Figura 9. Query Builder

Code Workbench

Code Workbench é um utilitário que permite habilitar e configurar alguns dos recursos providos pelo editor SQL da ferramenta. Sua interface é então mostrada na **Figura 10**, onde podem ser vistas três abas, cada qual com um direcionamento específico. A primeira, denominada Settings, apenas habilita/desabilita as opções disponíveis, enquanto que as outras duas fazem referência aos dois recursos relacionados ao editor SQL – Code Templates e Auto Replace – cuja finalidade principal gira em torno da busca pela agilidade e economia de tempo na codificação propriamente dita.

De forma conceitual, Code Templates podem ser definidos como sendo modelos pré-prontos de código, que são acionados via atalhos rápidos, durante o processo de codificação. Novamente traçando um paralelo com o cenário de desenvolvimento de aplicações, este recurso se assemelha

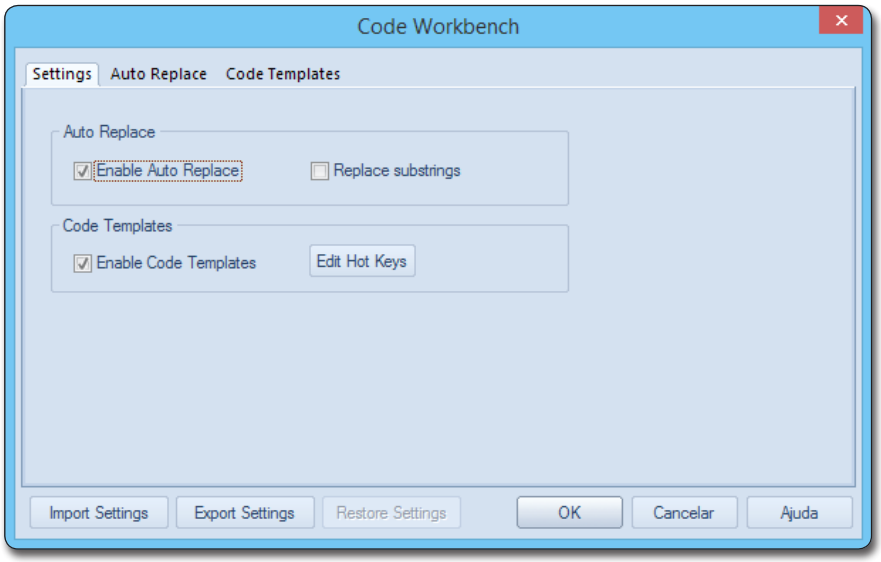


Figura 10. Code Workbench

ao Live Templates do Delphi, que permite o armazenamento local de "esqueletos" pré-elaborados de código. No Rapid SQL, cada template de código pode estar relacionado tanto a uma única plataforma de banco de dados ou a todas elas. A própria ferramenta já traz nativamente alguns Code Templates definidos. Como exemplo, a seguir é exposto o conteúdo de template nativo "If ElseIF Statement", voltado ao SBGB Oracle:

```
IF <condition> THEN
 <statements>;
ELSEIF <condition>
 <statements>;
ELSE <statements>;
END IF;
```

Em complemento, de forma similar, Auto Replace é um recurso que faz uso de um atalho ou expressão de substituição para inserir determinado trecho SQL no corpo do código que está sendo escrito. Novamente, o Rapid SQL já traz uma série de definições nativas, tal como mostrado na **Figura 11**, onde a expressão "cou", uma vez digitada, é substituída pelo comando SQL "select count(*) from".

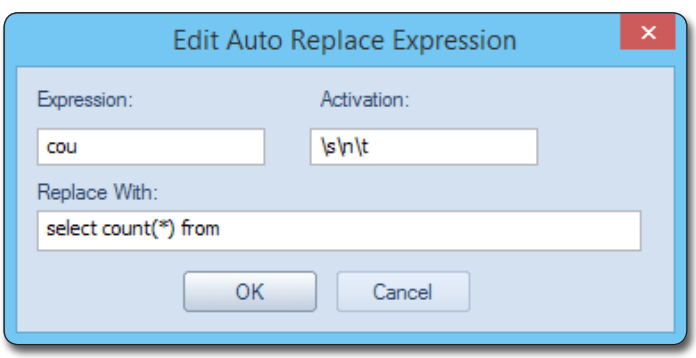


Figura 11. Edit Auto Replace Expression

Novidades do Rapid SQL XE5

Tradicionalmente, todo lançamento de uma nova versão de determinado produto acaba por envolver uma série de novos recursos, apresentados comercialmente como sendo suas "novidades". Sem fugir deste padrão, o novo release do Rapid SQL traz consigo uma gama de novos recursos e diversas melhorias, que podem ser segmentadas em três grupos principais, conforme expostos a seguir.

Melhorias gerais

Em termos gerais, o Rapid SQL XE5 apresenta ainda uma série de melhorias, mesmo que sutis, nos mais variados pontos de seu ambiente de desenvolvimento. Dentre estas, é possível citar seu utilitário Query Builder que agora atribui de forma automática um novo "alias" a cada instância de uma mesma tabela adicionada ao diagrama visual, útil nos casos de formação de "joins" recursivos por exemplo.

Rapid SQL para ambientes 64-bit e Windows 8

Até sua versão XE4, a instalação do Rapid SQL em ambientes Windows 64-bit se dava de forma emulada, uma vez que o produto tinha uma característica essencialmente 32-bit. Diante deste cenário, a fim de acompanhar a evolução do próprio meio ao qual está inserida, a ferramenta conta agora com uma versão nativa 64-bit. Logo, agora são dois instaladores disponíveis ao grande público, segmentados pela especificação da plataforma direcionada ao aplicativo – Rapid SQL XE5 32-bit e Rapid SQL XE5 64-bit. Como consequência, toda a parte de drivers "nativos" da ferramenta, usados no seu trabalho com os diversos SGBDs suportados, ganham também sua versão 64-bit e são devidamente instaladas nesta opção. Em complemento, o uso de drivers personalizados em substituição aos "nativos", agora exige a utilização da versão adequada destes, mediante ao tipo da plataforma envolvida, tal qual drivers 32-bit para Rapid SQL 32-bit e drivers 64-bit para Rapid SQL 64-bit.

Assim como grande parte dos produtos Embarcadero, o Rapid SQL é um software voltado exclusivamente ao sistema operacional Windows. Sendo assim, uma de suas premissas básicas é garantir a execução do aplicativo nas mais variadas edições desta plataforma. Até o Rapid SQL XE4, essa garantia se restringia até a versão 7 do SO que, até certo ponto, acabava por frustrar as intenções de seus usuários mais avançados, que faziam uso do título posterior a esta. Agora com a chegada do Rapid SQL XE5, este entrave é plenamente solucionado pela garantia do total suporte da ferramenta ao Windows 8.

Atualização do suporte a SGBDs

Nesta sua nova versão, o Rapid SQL apresenta uma atualização panorâmica de seu relacionamento direto com banco de dados,

representada contextualmente de formas distintas. A primeira delas se dá através do suporte a novas versões de determinados SGBDs, mais especificamente Oracle 12c, Sybase ASE 15.7, IBM DB2 LUW 10 e IBM DB2 z/OS 10. Adicionalmente, o Rapid SQL XE5 conta ainda com o suporte a um novo banco de dados, denominado Teradata, da empresa Teradata Corporation, em suas versões 13.10, 14 e 14.10. Para a conectividade com este SGBD, a ferramenta faz uso do Teradata JDBC driver, caso o respectivo driver ODBC não esteja instalado.

Ainda seguindo por esta linha, o Rapid SQL XE5 traz também novas funcionalidades relacionadas aos principais SGBDs líderes de mercado. Como exemplo, especificamente para bancos MS SQL Server 12 e posteriores, o software agora prevê o suporte a objetos do tipo Sequence, por meio de ações direcionadas e de elementos como editores e assistentes de criação. Na mesma medida, para bancos Oracle, o mecanismo interno da ferramenta agora é capaz de seguir automaticamente por rotas alternativas de conexão a uma fonte de dados, tal qual ao fazer uso do Oracle Database Instant Client em substituição ao Oracle SQL*Net Client (ver **BOX 3**), quando este não se encontra disponível no ambiente.

BOX 3. Oracle SQL*Net

De forma popular, o Oracle SQL*Net pode ser definido como sendo o software responsável por prover a comunicação, seja ela cliente-servidor ou servidor-servidor, entre bancos de dados Oracle e as aplicações clientes relacionadas. Logo, por padrão, ele é tido como a principal tecnologia de acesso remoto de dados da Oracle Corporation.

Interbase e Firebird podem ser considerados dois dos principais SGBDs presentes e atuantes no contexto Delphi. Sendo assim, a busca por um conhecimento técnico mais aprofundado de ambos, bem como a habilidade em utilizar ferramentas para sua manipulação, se tornam itens importantes e essenciais para a carreira do desenvolvedor Delphi. Em vista disso, o presente artigo trouxe como tema o Rapid SQL, uma ferramenta de banco de dados da Embarcadero, cuja sua mais nova versão, denominada XE5, estabelece ainda mais seus plenos recursos que visam atender aos mais variados tipos de banco de dados, incluindo os já citados InterBase e Firebird, além de Oracle, MySQL, DB2, MS SQL Server, entre outros. Conforme sua apresentação e a mostra das facilidades providas pelo seu uso, torna-se natural a estabilização do Rapid SQL XE5 como uma inteligente escolha, num mercado bastante concorrido, tal como é o de utilitários de banco de dados.

Por conseguinte, o uso de uma ferramenta auxiliar, como o Rapid SQL, oriunda da mesma fabricante de sua ferramenta principal de trabalho (Delphi), tende a proporcionar ao profissional diversos benefícios que vão desde a parte técnica, passando pelo conhecimento, até a parte comercial (de aquisições de licenças).

Autor

Fabrício Hissao Kawata
fabricio.kawata@bol.com.br

Formado em Processamento de Dados pela FATEC-TQ e pós-graduado em Engenharia de Componentes. Atua como Analista Programador Delphi há 9 anos. Atualmente escreve artigos sobre Delphi para revistas especializadas. Certified Delphi Developer. Certified Embarcadero Instructor. Site pessoal: www.fabriciokawata.com.br

Links:

Embarcadero Rapid SQLXE5 - Página oficial do produto
http://www.embarcadero.com/products/rapid-sql

Embarcadero Rapid SQL XE5 - Página de download
https://downloads.embarcadero.com/free/rapid_sql

Embarcadero InterBase XE3 - Página oficial do produto
https://www.embarcadero.com/products/interbase

Embarcadero InterBase XE3 - Página de download
https://www.embarcadero.com/products/interbase/downloads

Firebird - Página oficial do projeto
http://www.firebirdsql.org

Firebird - Página de download
http://www.firebirdsql.org/en/downloads

Firebird – Driver ODBC
http://www.firebirdsql.org/en/odbc-driver/

Você gostou deste artigo?

Dê seu voto em ***www.devmedia.com.br/clubedelphi/feedback***

Ajude-nos a manter a qualidade da revista!

TMS Data Modeler: Desenvolva um projeto de Banco de Dados

Tenha o controle

De forma sucinta, o TMS Data Modeler pode ser definido como sendo uma ferramenta visual de banco de dados, de cunho comercial, produzida pela empresa TMS Software. Sua principal característica está associada à sua usabilidade, provida através de uma interface visual clara, intuitiva e eficiente o bastante para as tratativas que envolvem os mais diferentes tipos de bancos de dados suportados, tais como Oracle, Microsoft SQL Server, MySQL e Firebird. Em outras palavras, o TMS Data Modeler integra num único ambiente, soluções para as questões de modelagem, criação e manutenção de banco de dados. Já neste primeiro momento vale destacar um aspecto peculiar da ferramenta, que é o seu trabalho com "projetos de banco de dados". Sendo assim, para que as tratativas aconteçam, é invariavelmente necessária a criação prévia de um Projeto.

Download e instalação

O uso do TMS Data Modeler envolve a aquisição de uma licença de uso comercial para um (Single Developer License) ou dois desenvolvedores (Small Team License). Todavia, assim como a grande maioria das ferramentas pagas, é disponibilizada de forma gratuita (ver seção **Links**), numa vigência de 30 dias, uma versão Trial de avaliação. Para este trabalho, será tomada como base a versão 2.2.3.1 (**Figura 1**), que se mostrava a mais atual até a data da escrita deste artigo. Em termos de instalação, o produto segue o tradicional padrão Windows (Next-Next-Finish), dispensando qualquer comentário adicional. Já em sua primeira execução, um pequeno lembrete é mostrado, mencionando o tipo de distribuição do produto (shareware), bem como os dias restantes de avaliação (**Figura 2**).

Fique por Dentro

O desenvolvimento de aplicações de banco de dados no Delphi é sem sombra de dúvidas um de seus grandes pilares de excelência. Aliado a isso, a história nos mostra o Firebird como sendo uma das principais escolhas em SGDB para este tipo de aplicação. Por esta razão, conhecê-lo se torna algo essencial para a efetiva construção de tais projetos, o que acaba por envolver sua plena criação e manutenção. Em vista disso, este artigo apresenta o TMS Data Modeler, ferramenta voltada à sustentabilidade de diversos tipos de bancos de dados diferentes, incluindo o Firebird.

Recursos

Dentre os inúmeros recursos trazidos pelo TMS Data Modeler, alguns deles podem ser destacados, conforme listados a seguir:

- Usabilidade facilitada através de uma interface de usuário simples e intuitiva;
- Suporte otimizado aos principais SGBDs do mercado, os quais são condizentes ao cenário Delphi, tais como: Firebird, Oracle, MS SQL Server, SQLite e MySQL, além do serviço de banco de dados na nuvem da Microsoft, o SQL Azure;
- Agilidade no processo de leitura e escrita das atividades desempenhadas, uma vez que lida com um formato de arquivo próprio (.dgp);
- Produção de scripts SQL, tanto para a criação de um banco de dados novo quanto para atualização de uma base existente;
- Engenharia reversa, para a visualização e ajustes na estrutura de um banco de dados efetivo;
- Controle de versão do banco de dados, provendo recursos suficientes para mapear e acompanhar as mudanças surgidas ao longo do tempo;
- Recursos para modelagem: criação de DER (Diagrama Entidade Relacionamento);
- Conversão de banco de dados para diferentes SGBDs.

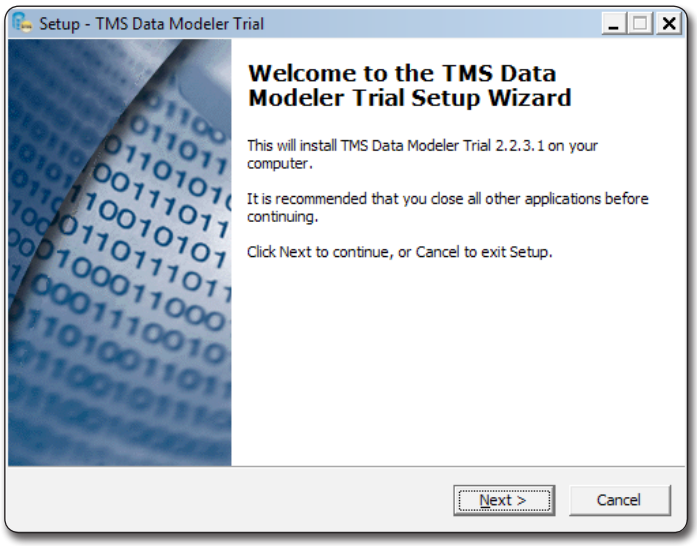


Figura 1. TMS Data Modeler – tela inicial de instalação

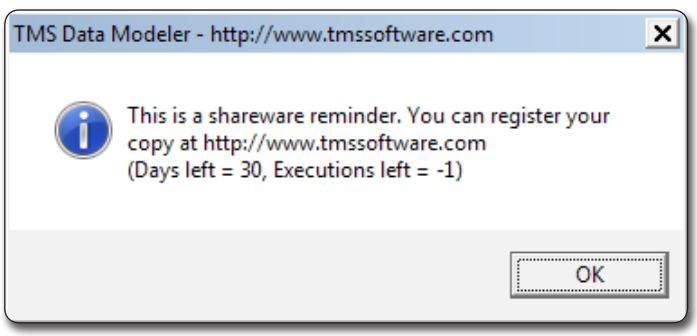


Figura 2. TMS Data Modeler – mensagem inicial de execução

Interface

A interface de usuário do TMS Data Modeler adota o padrão Ribbon, tal como mostra a **Figura 3**, que é um estilo de interface introduzido pelo Microsoft Office 2007, cuja intenção é melhorar ainda mais a experiência do usuário, através do uso de elementos mais representativos. Prova disso, são os botões mostrados na aba Home, que ficam dispostos em grupos de interesse, tais como Project, Create, Versions e Show/Hide.

Sem deixar de mencionar, a seguir são descritas algumas das funcionalidades providas pela ferramenta através destes botões:

- Generate Script: a partir de uma prévia seleção dos tipos de objetos pertencentes ao banco de dados em questão, produz um script SQL de criação dos mesmos;
- Check: realiza a validação do Projeto, verificando questões de campos, relacionamentos e chaves;
- Merge: após um prévio comparativo, realiza a fusão entre dois projetos distintos em um só, gerando os scripts necessários;
- Table: determina a criação de uma nova tabela;
- Relationship: determina a criação de um novo relacionamento entre tabelas;
- Domains: mostra os domínios existentes, possibilitando editá-los, bem como a criação de novos.

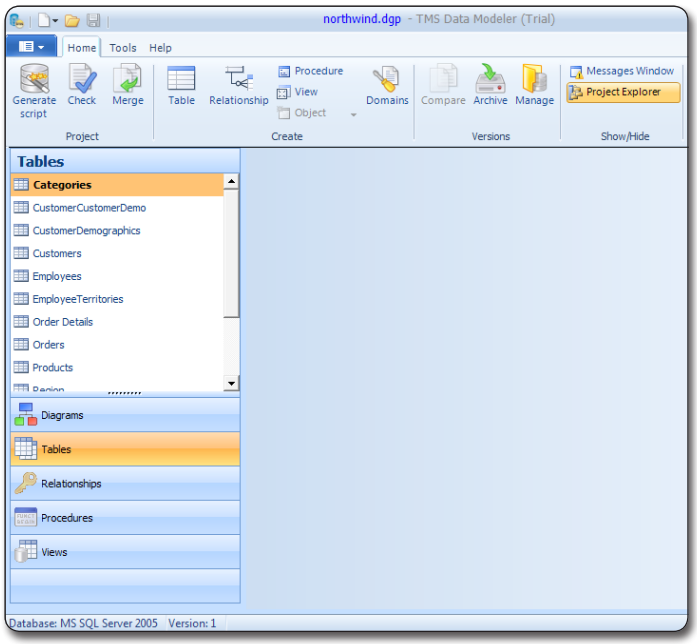


Figura 3. Interface do TMS Data Modeler

Project Explorer

No IDE do Delphi, o Project Manager é um elemento de interface que provê, entre outras coisas, a visualização dos itens que compõem um projeto. De forma similar, na interface do TMS Data Modeler, o Project Explorer é quem faz esta função, conforme pode ser visto na **Figura 4**. Sendo assim, através dele é possível se visualizar os membros do projeto, tais como os objetos de banco de dados (tabelas, procedures, views, generators/sequences) além de relacionamentos e diagramas. Ainda mediante o comparativo estabelecido, da mesma forma que um duplo-clique sobre um item .pas de um projeto VCL Forms no Delphi faz com que a unit seja aberta no Form Designer/Code Editor, aqui o duplo-clique sobre um dos elementos de projeto do TMS Data Modeler faz com que o objeto seja aberto na área principal da ferramenta, conhecida como Workspace.

Workspace

A Workspace pode ser considerada a principal parte integrante da interface do TMS Data Modeler, uma vez que, por padrão, ocupa toda a porção central da ferramenta. Em vista disso, ele é tido como a "área de trabalho" da mesma, onde ficarão dispostas em abas as informações manipuláveis do projeto. Exemplificando este cenário, a **Figura 5** exibe a Workspace com diversos objetos "Tabela" abertos.

Messages Window

Messages Window nada mais é do que uma simples caixa de texto localizada na parte inferior da interface do TMS Data Modeler. No contexto do desenvolvimento, ela é tida como uma janela de mensagens, uma vez que é responsável por exibir as eventuais notificações sobre determinado projeto.

Por padrão, ela vem desabilitada, sendo necessário o acionamento do botão de mesmo nome localizado no agrupamento Show/Hide da guia Home, para que se torne visível.

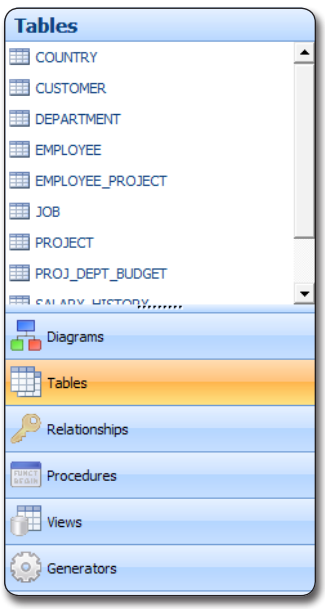


Figura 4. Project Explorer

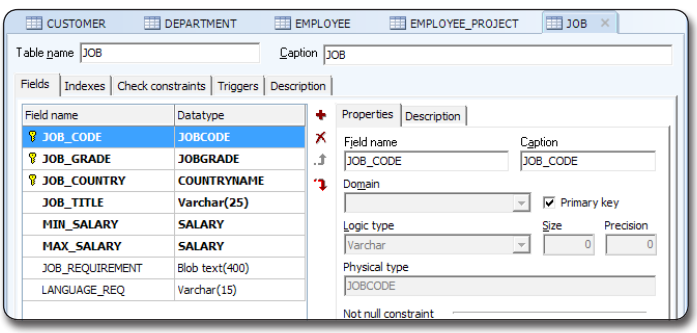


Figura 5. Workspace

Modelo de banco de dados

Nesta ferramenta da TMS Software, o ponto de partida natural de seu uso se reflete na criação de um novo projeto. Para isso, duas são as opções disponíveis, que é a definição de um projeto em branco ou baseado em um banco de dados já existente. No caso específico do primeiro, uma das práticas disponíveis envolve a definição de um modelo de banco de dados, que na interface do TMS Data Modeler é apresentado como um diagrama. Diferente de outras ferramentas exclusivas de modelagem, que trabalham com modelos "genéricos", ou mesmo com segmentações que envolvem modelos conceituais, lógicos e físicos, o TMS Data Modeler considera apenas uma modelagem única, do tipo Entidade-Relacionamento. Sendo assim, já no momento inicial da atividade (**Figura 6**) se faz necessária a indicação explícita do SGBD relacionado (de destino). A escolha entre determinado tipo de banco de dados será então refletido no contexto das definições do modelo em si, tais como objetos, tipos de dados, relacionamentos e, principalmente, no script a ser gerado pela ferramenta.

Diagrama

O início de um novo projeto "em branco" no TMS Data Modeler não está essencialmente atrelado à definição de um modelo de dados, sendo possível também a simples definição da estrutura do banco de dados sem o uso de um diagrama. Todavia, a utilização de uma estrutura como esta tende a proporcionar diversos benefícios, nos mais variados aspectos, muito em razão de seu apelo visual. Conceitualmente falando, no contexto desta ferramenta da TMS, um diagrama atua como uma visão do projeto, podendo englobar tanto uma parte quanto o seu todo. Sendo assim, na prática, é plenamente viável a criação de vários diagramas para um mesmo projeto, cada qual abrangendo determinadas partes da estrutura. De forma ilustrativa, a **Figura 7** mostra o diagrama de exemplo que acompanha a instalação do TMS Data Modeler, relativo ao banco Northwind do Microsoft SQL Server.

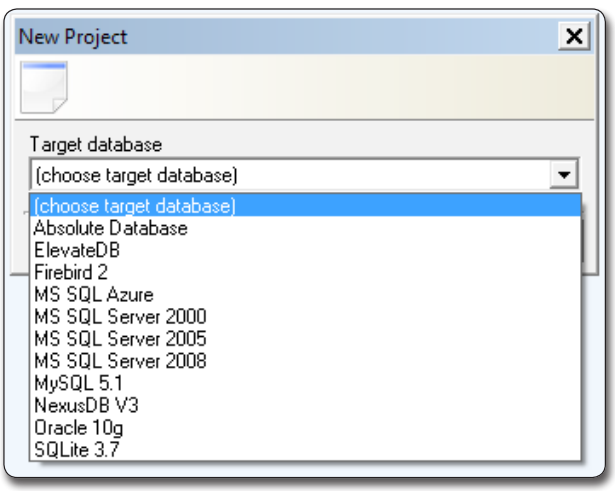


Figura 6. Novo projeto – indicando o SGBD de destino

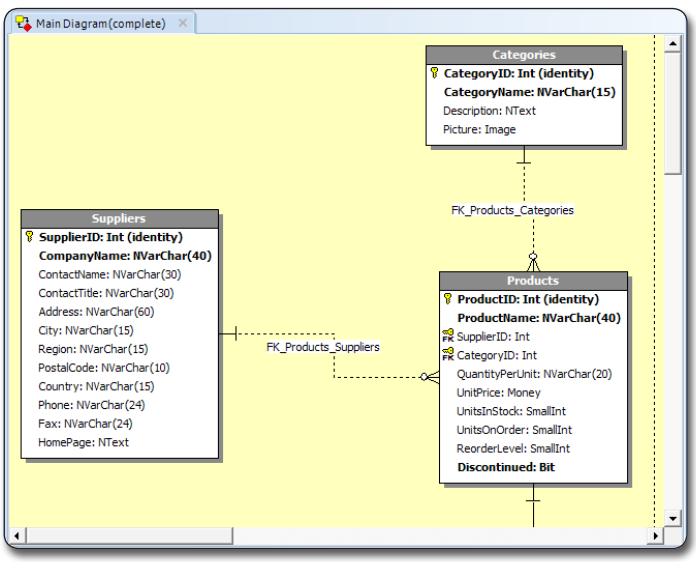


Figura 7. Diagrama no TMS Data Modeler

Em algumas ferramentas concorrentes do gênero, o conceito adotado para diagramas se mostra um tanto quanto diferente, uma vez que ele é considerado peça única para todo o projeto. Nestes casos, toda a definição da estrutura do banco de dados se concentra essencialmente em um digrama único e central.

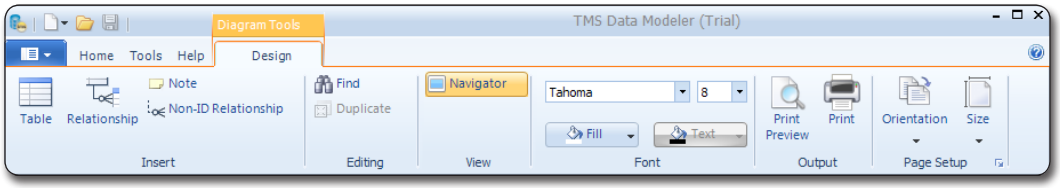


Figura 8. Interface do TMS Data Modeler – guia Design

Ferramentas de Design

De posse do diagrama, a efetiva modelagem do banco de dados se inicia, como já é de praxe, pela definição dos objetos que irão compor a base. Em vista disso, o TMS Data Modeler apresenta em sua interface, de forma exclusiva ao trabalho com diagramas, uma guia de opções denominada Design (**Figura 8**), que contempla todas as funcionalidades necessárias para teste tipo de atividade, incluindo a criação de objetos.

Tabela e Campos

Logo, novamente tomando como base o que é tradicional, o primeiro passo para a construção de um modelo de banco de dados é a definição de suas tabelas. Isto porque, sob a ótica da grande maioria das ferramentas do gênero, os objetos do tipo Table são considerados os elementos mais básicos de um projeto de banco de dados. No IDE do TMS Data Modeler, isso é conseguido através do botão "Table", mostrado na imagem anterior. Seu simples acionamento em direção ao diagrama faz com que um objeto tabela seja adicionado visualmente ao modelo, bem como um editor exclusivo (Table Editor) seja imediatamente aberto na Workspace. A **Figura 9** mostra então o Table Editor já previamente preenchido. Conforme pode ser observado na imagem, além da tabela propriamente dita, o que inclui seus campos, o editor trabalha com as definições de outros elementos associados a ela, tais como restrições (constraints e check constraints), índices (indexes) e gatilhos (triggers).

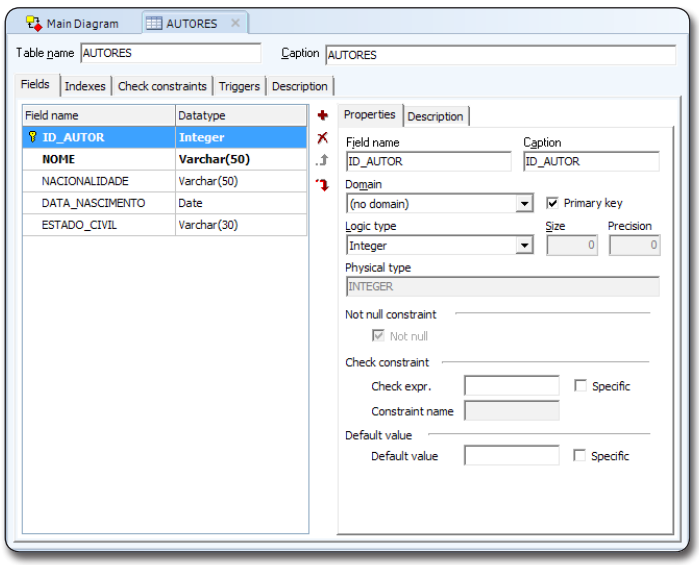


Figura 9. Editor de tabela

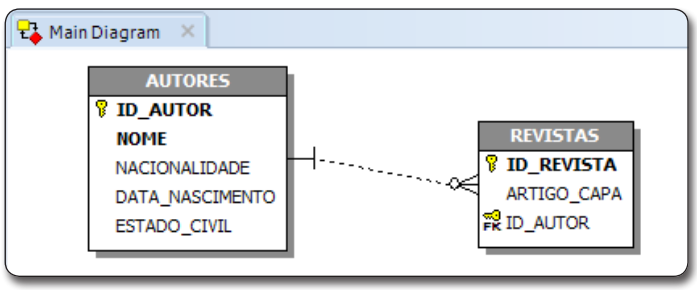


Figura 10. Relacionamento sendo mostrado no diagrama

Relacionamentos

No contexto de banco de dados, um relacionamento nada mais é do que uma ligação entre tabelas. Num exemplo simbólico, uma tabela de Pessoas pode se relacionar com outra de Países através de um campo ID_PAIS presente em sua estrutura. Logo, este campo estaria "ligado" ao campo Chave Primária (ex: ID_PAIS) da tabela de Países. Em vista disso, o TMS Data Modeler exige o envolvimento de um campo definido como chave para a efetivação de relacionamentos. Além disso, a ferramenta trabalha com dois tipos de relacionamentos distintos:

- Identificado (Identifying relationship): caracteriza uma ligação "forte", em que os campos da Tabela "A" que estão relacionados à Tabela "B", fazem parte da composição de sua chave. Em outras palavras, tais campos se incluem na identificação única de cada registro presente na Tabela "A";
- Não-Identificado (Non-Identifying relationship): caracteriza uma ligação "fraca". Pode ser considerado o tipo mais comum de relacionamento, onde o campo da Tabela "A" que se relaciona com a Tabela "B" não faz parte da composição de chave.

Ainda tendo como base o diagrama, a criação de um relacionamento se dá a partir do botão Relationship da guia Design. Ao se definir uma relação, esta é mostrada no designer do diagrama sob a forma de uma linha contínua que liga as entidades envolvidas, tal como pode ser visto na **Figura 10**.

Domínios

Em banco de dados, um domínio (ou Domain), é um recurso que permite a definição de um tipo especial de campo. Uma vez criada, esta definição pode então ser reutilizada por entre vários campos do banco de dados, caracterizando um "padrão". Numa exemplificação simples, tomando como base o banco Employee do Firebird, este já traz consigo diversos Domains criados. Dentre estes, PHONENUMBER é o nome do domínio que caracteriza um

tipo de campo de número de telefone. A seguir são mostrados os detalhes de sua definição:

```
CREATE DOMAIN PHONENUMBER AS VARCHAR(20);
```

Logo, esta mesma definição, ou seja, este mesmo domínio é utilizado em diversos campos do banco de dados, tal como em PHONE_NO da tabela DEPARTMENT, conforme mostrado a seguir:

```
CREATE TABLE DEPARTMENT (
 DEPT_NO DEPTNO NOT NULL,
 DEPARTMENT VARCHAR(25) NOT NULL,
 HEAD_DEPT DEPTNO,
 MNGR_NO EMPNO,
 BUDGET BUDGET,
 LOCATION VARCHAR(15),
 PHONE_NO PHONENUMBER,
 CONSTRAINT INTEG_16 PRIMARY KEY (DEPT_NO)
);
```

Sendo assim, na prática, o domínio atua como um tipo de dado. Nos bastidores, de forma implícita, o campo acaba por assumir a definição do Domain, ou seja, VARCHAR (20). No TMS Data Modeler, toda a definição de domínios se concentra no botão Domains, que dá origem a uma janela homônima (**Figura 11**).

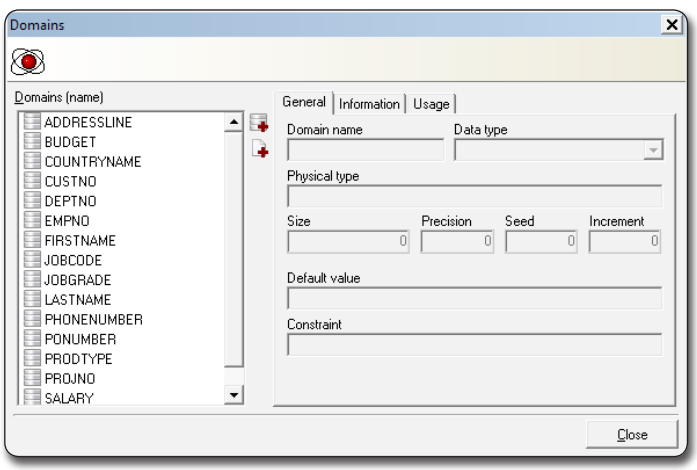


Figura 11. Janela Domains

Além disso, vale ressaltar que a ferramenta da TMS trabalha com dois tipos de domínios: logico e físico. Na prática, o tipo "logico" caracteriza um domínio utilizado apenas em tempo de projeto, não sendo criado fisicamente no banco de dados. Sendo assim, o campo é gerado com as mesmas atribuições do domínio utilizado, porém sem uma menção direta a ele. Já um domínio físico, conforme seu nome sugere, é efetivamente gerado como um objeto do banco de dados.

Geração de scripts SQL

No TMS Data Modeler, a posse de um projeto de banco de dados possibilita tanto a gerência de sua estrutura, quanto a produção de scripts SQL relacionados. Para este último, a ação principal provida pela ferramenta se concentra no botão Generate Script, já mencionado anteriormente. Por meio dele, entra em cena a janela Script Generation, mostrada na **Figura 12**, que possibilita a seleção específica dos tipos de objetos que estarão envolvidos no processo, além dos próprios objetos nomeados. Tomando como base o uso de diagramas, tal qual o mostrado na imagem anterior, o script produzido para sua estrutura pode ser vista na **Listagem 1**.

Listagem 1. Script gerado a partir da estrutura mostrada na **Figura 12**

```
CREATE TABLE REVISTAS (
 ID_REVISTA INTEGER NOT NULL,
 ARTIGO_CAPA VARCHAR(50),
 ID_AUTOR INTEGER,
 CONSTRAINT PrimaryKey3 PRIMARY KEY (ID_REVISTA)
);

CREATE TABLE AUTORES (
 ID_AUTOR INTEGER NOT NULL,
 NOME VARCHAR(50) NOT NULL,
 NACIONALIDADE VARCHAR(50),
 DATA_NASCIMENTO DATE,
 ESTADO_CIVIL VARCHAR(30),
 CONSTRAINT PrimaryKey1 PRIMARY KEY (ID_AUTOR)
);

ALTER TABLE REVISTAS ADD CONSTRAINT AUTORES_REVISTAS
 FOREIGN KEY (ID_AUTOR)
 REFERENCES AUTORES (ID_AUTOR);
```

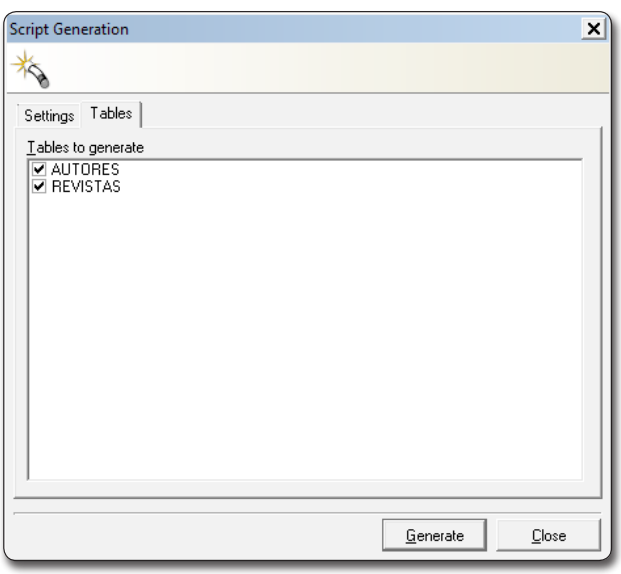


Figura 12. Janela Script Generation

Análise de consistência de um projeto

Dentre seus diversos recursos, o TMS Data Modeler conta com um utilitário que verifica a consistência de um projeto, abrangendo todos os seus itens e configurações. Tal validação é realizada por meio do acionamento do botão Check, que no mesmo instante dá início ao processo. Internamente, esta operação trabalha em conjunto com o painel de mensagens da ferramenta (Messages Window), uma vez que toda e qualquer notificação produzida pela

ação Check, relacionada ao projeto, será direcionada a ela. Como definição, tais notificações são mensagens que indicam eventuais inconsistências que possam afetar diretamente a geração da estrutura do banco de dados. Neste cenário, estas mensagens são classificadas em dois tipos: Mensagem de erro (Error Message) e Mensagem de aviso (Warning Message). O primeiro tipo faz referência a erros concretos detectados nas configurações de objetos do projeto enquanto que o segundo faz a possíveis erros, servindo como um alerta ao usuário. A seguir são listados alguns exemplos de mensagens produzidas neste processo:

Errors Messages

• "Duplicate field name [nome_do_campo] on table [nome_da_tabela]": mensagem de erro produzida quando da existência de nomes duplicados para campos em uma determinada tabela;
• "Duplicate table name [nome_da_tabela]": mensagem de erro produzida quando da existência de nomes duplicados para tabelas, na estrutura de banco de dados;
• "Field has no name on table [nome_da_tabela]": mensagem de erro gerada quando é detectada a presença de campos não nomeados na tabela;
• "Relationship has no name": mensagem de erro que aponta a presença de um relacionamento entre tabelas não nomeadas, na estrutura do projeto;
• "Table has no name": mensagem de erro que aponta a presença de uma tabela não nomeada na estrutura do projeto.

Warning Messages

• "Constraint name [nome_da_constraint] on table [nome_da_tabela] is a reserved word": mensagem de aviso produzida quando o nome de uma Constraint de uma tabela faz menção a alguma palavra reservada da linguagem SQL do SGBD em questão;
• "Table [nome_da_tabela] has no fields": mensagem de aviso que alerta o usuário sobre a presença de uma determinada tabela na estrutura de banco de dados, que não contém qualquer definição de campos (fields);
• "Name too long for field [nome_do_campo] on table [nome_da_tabela]. Maximum size is [tamanho] characters": mensagem de aviso que alerta sobre um nome de campo em determinada tabela que ultrapassa o tamanho máximo permitido de caracteres;
• "Size was not specified on field [nome_do_campo]": mensagem de aviso que alerta sobre a presença de um campo de tabela que não apresenta especificação de tamanho;
• "Table [nome_da_tabela] has no primary key": mensagem de aviso que informa a presença de determinada tabela sem a definição de chave primária (primary key);

Configurações de projeto

A guia de opções Tools do TMS Data Modeler traz em seu agrupamento Project o botão Settings, que é responsável pelas configurações gerais do projeto. Na janela que se abre, a ferramenta provê então duas guias informativas (Information e Version Control), onde o usuário poderá prover dados sobre determinados aspectos do projeto corrente, tais como nome do projeto (Project name), nome do autor (Author), descrição (Description), bem como a especificação do local de armazenamento dos arquivos de controle de versão. Para este último, por padrão, é definida uma pasta "versions", dentro do próprio diretório do projeto. A fim de ilustrar tudo isso, a **Figura 13** mostra a janela Settings com uma configuração prévia.

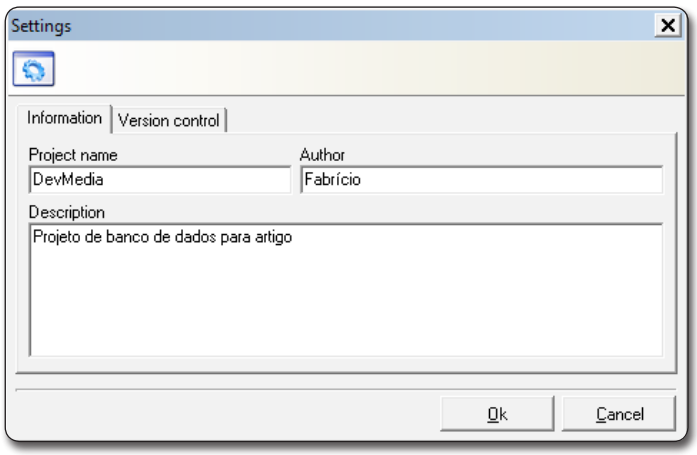


Figura 13. Configurações gerais do projeto

Engenharia reversa

Em tese, o fluxo normal de operações do TMS Data Modeler envolve a criação efetiva de um banco de dados no mesmo instante da definição de sua estrutura, a partir de um "projeto de banco de dados". De forma complementar, a ferramenta é capaz ainda de operar no sentido reverso deste fluxo, nas ocasiões onde se tem a posse de um banco de dados já elaborado anteriormente, e se almeja o acesso à sua estrutura (tanto para visualização quanto para modificação).

No IDE, esta engenharia reversa é provida por meio da opção de menu New | Import from Database. Ao acioná-la, assim como ocorre no Delphi, um assistente (Wizard) é aberto, com o intuito de guiar o usuário pelos ajustes necessários ao processo, através de seus passos. Tomando como base o SGBD Firebird e o banco de dados de exemplo "employee.fdb" que acompanha sua instalação, a seguir nas **Figuras 14** e **15** são mostrados os passos que envolvem esta definição. Em suma, numa primeira etapa, é definido o tipo do SGBD do arquivo de banco de dados em questão, e posteriormente, as configurações do mesmo, tais como as credenciais de acesso (usuário e senha), a localização física do arquivo, o servidor utilizado e o protocolo de comunicação. Ao final das etapas e, consequentemente, do processo, a engenharia reversa é então efetivada. Isso é comprovado pela imediata visualização da estrutura do banco de dados na interface da ferramenta.

Controle de versão

Conforme já mencionado, um dos recursos providos pelo TMS Data Modeler é um eficiente controle de versão embutido no IDE que, diferentemente de outras interfaces, não exige qualquer

instalação adicional de plug-ins para tal função. Toda esta simplicidade oferecida pela ferramenta é fruto de sua eficiente arquitetura que persiste toda a estrutura do banco de dados em um arquivo de extensão própria (.dgp). Logo, todo seu controle de versionamento se dá única e exclusivamente sobre este arquivo.

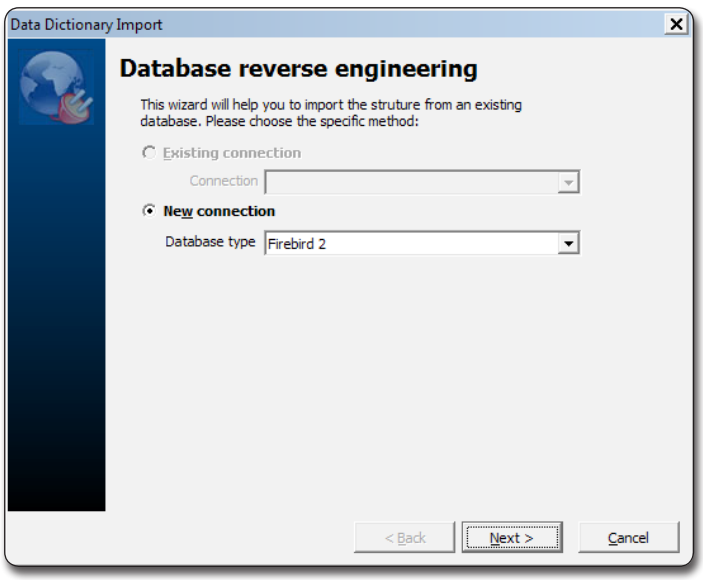


Figura 14. Engenharia reversa – Wizard – selecionando o SGBD

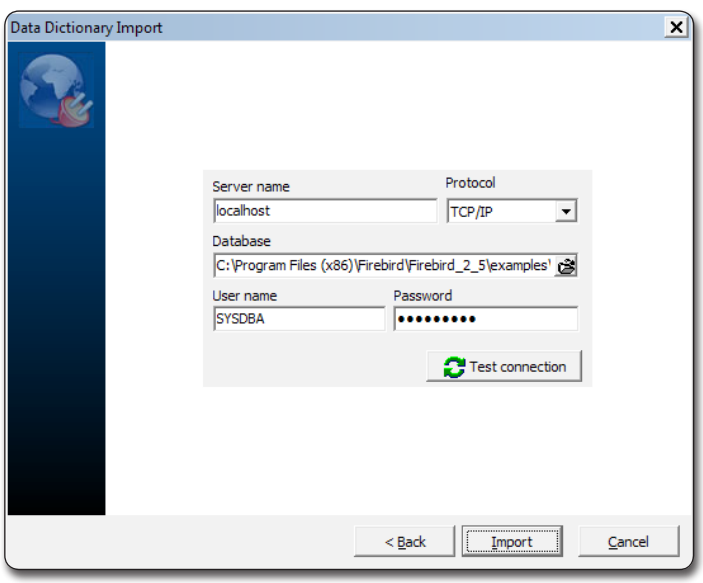


Figura 15. Engenharia reversa – Wizard – configurações do banco de dados

Com isto em mente, é possível se estabelecer a ação de salvar (Save) como passo fundamental para a realização do controle de versão em determinado projeto de banco de dados no TMS Data Modeler. Tomando como base a exemplificação anterior, onde um novo projeto foi criado a partir de uma base de dados existente, este ainda não está apto a ser versionado, uma vez que ainda não foi persistido em disco. No IDE, as ações relacionadas a controle de versão ficam localizadas no agrupamento denominado Versions, tal como mostra a **Figura 16**. Dentre estas, a opção Archive é a que dá início ao versionamento e permanece inativa enquanto o projeto não for efetivamente salvo.

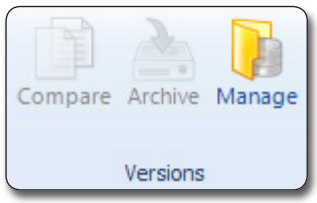


Figura 16. Controle de versão – opção Archive desativada

Uma vez habilitada, ao acionar tal opção, uma caixa de diálogo denominada Archive Version é aberta. Sua utilização tem a finalidade de demarcar a versão que está sendo trabalhada naquele exato momento. Em vista disso, sua interface apresenta de forma dinâmica e automática os informativos do número da versão que está sendo estabelecida, bem como a data e a hora corrente. Adicionalmente, dá abertura também para que o usuário forneça qualquer informação relevante que identifique a versão atual. No TMS Data Modeler, o final de versionamento é marcado por uma mensagem informativa (**Figura 17**) que contempla a adição da versão salva ao controle interno, seguido do novo número de versão em uso.

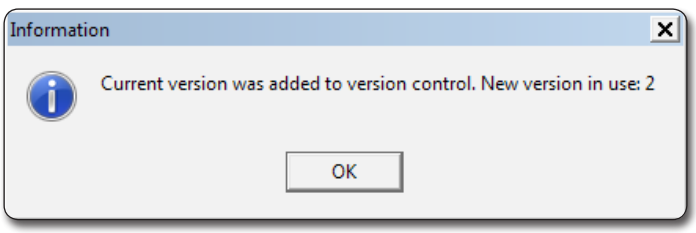


Figura 17. Controle de versão – mensagem de sucesso de versionamento

Do ponto de vista do cotidiano de manutenção da estrutura de um banco de dados no TMS Data Modeler, a existência de diferentes versões do projeto significa a possibilidade de visualizar e restaurar uma versão anterior a qualquer momento. Na interface da ferramenta, tais operações ficam centralizadas no botão Manage do agrupamento Versions. Quando acionado, uma nova janela interativa é aberta, denominada Project Versions (**Figura 18**), onde são então listadas as versões salvas, bem como as ações relacionadas.

Em complemento, o botão Compare, conforme seu nome sugere, estabelece um comparativo entre versões. Esta ação trabalha essencialmente com dois momentos distintos, caracterizando um Wizard. Sendo assim, numa primeira etapa, as versões a serem comparadas devem ser informadas e, posteriormente, o resultado deste comparativo é exibido em tela. Ainda na primeira etapa, duas são as situações as quais a ferramenta trabalha e dizem respeito à existência ou não de diferenças entre as versões comparadas. Para este último caso, a própria ferramenta se encarrega de notificar o usuário, evitando que o mesmo dispenda tempo desnecessário ao processo.

Por fim, a **Figura 19** mostra a janela de resultados da comparação entre versões, onde são apontadas, tanto de forma visual quanto no formato de script, as diferenças existentes. O destaque neste cenário fica por conta do botão Generate que, numa ação conjunta com a opção Action, marcada como Generate Database Script, irá produzir os scripts de atualização necessários para a resolução das diferenças existentes entre as estruturas comparadas. Tomando como base o exemplo mostrado na imagem, onde entre uma versão e outra foi adicionado um novo campo (TESTE) na tabela COUNTRY do banco Employee do Firebird, o script gerado pela ferramenta para a resolução desta diferença é mostrado a seguir:

```
ALTER TABLE COUNTRY ADD TESTE INTEGER;
```

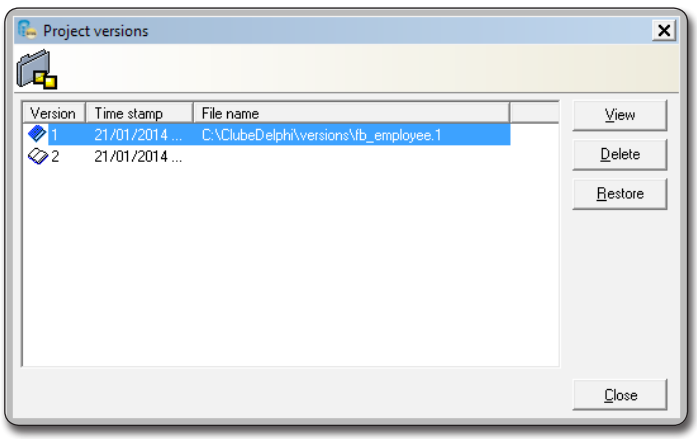


Figura 18. Controle de versão – versões do projeto

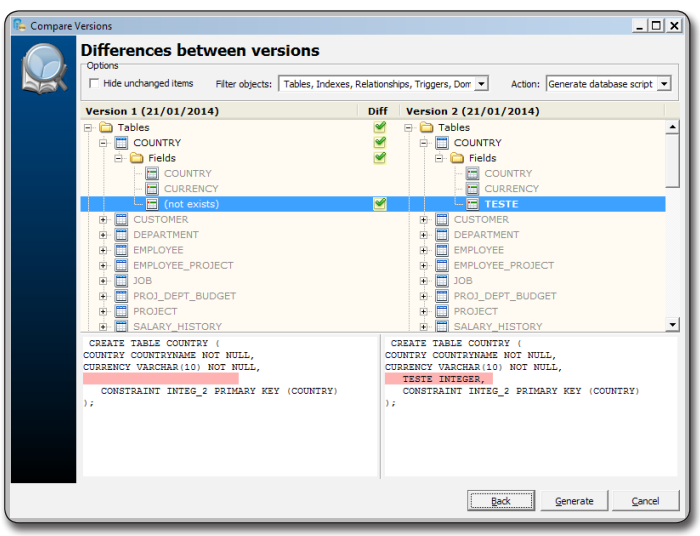


Figura 19. Controle de versão – comparação de versões

Conversão entre bancos de dados

Conforme já mencionado, o TMS Data Modeler contempla o recurso de conversão entre bancos de dados. Na prática, esta ação permite a troca do tipo de banco de dados, ou seja, que a estrutura de uma base de dados relativa a um SGBD "X" seja direcionada a outro SGBD "Y", tal como de Firebird para Oracle, de MS SQL Server para Firebird e assim por diante. O início deste processo se dá pelo acionamento do botão Convert, localizado no agrupamento Project da guia Tools. Logo, a caixa de diálogo Convert Database (**Figura 20**) é aberta, onde deverão ser informados os tipos de bancos de dados envolvidos (corrente e novo).

Todo este processo incide, por exemplo, sobre o script de banco de dados que é gerado a partir da estrutura do projeto. Sendo assim, a **Listagem 2** mostra um trecho do script produzido pela ferramenta para os objetos (do tipo Tabela) "Project" e "Job" do banco Employee do Firebird. Já a **Listagem 3** exibe o mesmo script, mas um momento após a realização da conversão de Firebird para MS SQL Server.

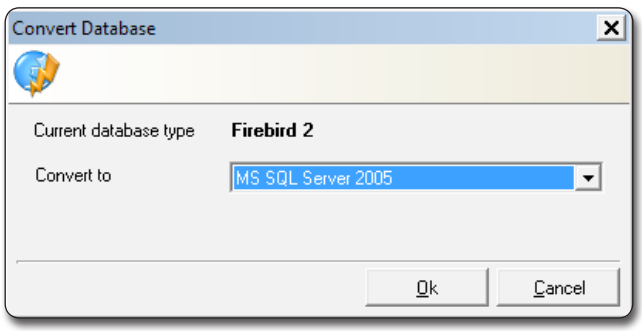


Figura 20. Convert Database

Listagem 2. Banco Employee – Script gerado pelo TMS Data Modeler para Firebird

```
CREATE TABLE PROJECT (
  PROJ_ID PROJNO NOT NULL,
  PROJ_NAME VARCHAR(20) NOT NULL,
  PROJ_DESC BLOB SUB_TYPE TEXT SEGMENT SIZE 80,
  TEAM_LEADER EMPNO,
  PRODUCT PRODTYPE,
  CONSTRAINT INTEG_35 PRIMARY KEY (PROJ_ID)
);

CREATE TABLE JOB (
  JOB_CODE JOBCODE NOT NULL,
  JOB_GRADE JOBGRADE NOT NULL,
  JOB_COUNTRY COUNTRYNAME NOT NULL,
  JOB_TITLE VARCHAR(25) NOT NULL,
  MIN_SALARY SALARY NOT NULL,
  MAX_SALARY SALARY NOT NULL,
  JOB_REQUIREMENT BLOB SUB_TYPE TEXT SEGMENT SIZE 80,
  LANGUAGE_REQ VARCHAR(15),
  CONSTRAINT INTEG_10 PRIMARY KEY (JOB_CODE,JOB_GRADE,JOB_COUNTRY),
  CONSTRAINT INTEG_12 CHECK (min_salary < max_salary)
);
```

Mapeamento de conversão

Apesar de à primeira vista parecer um processo natural, a conversão entre banco de dados necessita invariavelmente de outro processo, apresentado no contexto do TMS Data Modeler como Field Mapping. Numa tradução livre para o português, este processo pode ser chamado de Mapeamento de Campo, e caracteriza a seleção de equivalências entre os tipos de dados dos diferentes SGBDs. Isso se torna necessário pelo fato de que cada SGBD apresenta suas próprias nuances, no que diz respeito aos tipos de dados suportados, podendo inclusive disponibilizar

tipos mais específicos, tal como o tipo Clob para Oracle. Sendo assim, essa ação é a garantia para que não haja perda de dados durante o processo de conversão, por motivos quaisquer de incompatibilidade de dados.

Listagem 3. Banco Employee – Script gerado pelo TMS Data Modeler para MS SQL Server

```
CREATE TABLE [PROJECT] (
  [PROJ_ID] PROJNO NOT NULL,
  [PROJ_NAME] VARCHAR(20) NOT NULL,
  [PROJ_DESC] TEXT NULL,
  [TEAM_LEADER] EMPNO NULL,
  [PRODUCT] PRODTYPE NULL,
  CONSTRAINT [INTEG_35] PRIMARY KEY ([PROJ_ID])
)
GO

CREATE TABLE [JOB] (
  [JOB_CODE] JOBCODE NOT NULL,
  [JOB_GRADE] JOBGRADE NOT NULL,
  [JOB_COUNTRY] COUNTRYNAME NOT NULL,
  [JOB_TITLE] VARCHAR(25) NOT NULL,
  [MIN_SALARY] SALARY NOT NULL,
  [MAX_SALARY] SALARY NOT NULL,
  [JOB_REQUIREMENT] TEXT NULL,
  [LANGUAGE_REQ] VARCHAR(15) NULL,
  CONSTRAINT [INTEG_10] PRIMARY KEY ([JOB_CODE],[JOB_GRADE],
  [JOB_COUNTRY]),
  CONSTRAINT [INTEG_12] CHECK (min_salary < max_salary)
)
GO
```

Por padrão, o processo de conversão entre bancos do TMS Data Modeler se aproveita de um mapeamento implícito provido pela ferramenta, contudo, em vista da segurança prevista, a recomendação é pela realização do Field Mapping numa etapa anterior à conversão propriamente dita, independente da escala da base de dados envolvida. No IDE, esta opção de m go contendo a lista dos mapeamentos existentes, bem como as opções de adicionar um novo, alterar ou remover um existente. Durante a adição, uma nova janela é apresentada ao usuário sob o nome de Data Type Conversion Map, tal como mostra a **Figura 21**. Sua função, conforme sua nomenclatura sugere, é prover um mapa de conversão entre os tipos de dados dos SGBDs envolvidos (de origem e destino).

Mesclando projetos

O TMS Data Modeler conta ainda com um recurso de Merge, capaz de realizar a combinação entre dois projetos de banco de dados distintos, porém de um mesmo tipo de SGBD. A finalidade disso é fazer com que o projeto base (que está aberto na ferramenta) fique idêntico ao projeto que está sendo comparado. Para isso, a ferramenta realiza previamente uma comparação entre os projetos, expondo suas diferenças e, posteriormente, gerando um script para a resolução destas. No IDE, este recurso é provido por meio do botão Merge, do agrupamento Project, localizado na aba de opções Home.

Sob a perspectiva da realidade do desenvolvimento de software, é notória a atuação do profissional programador nas atividades

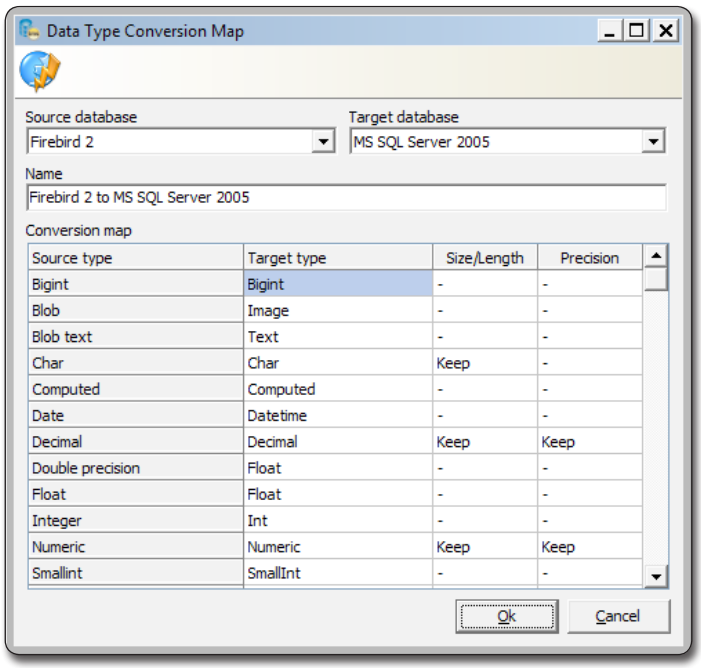


Figura 21. Data Type Conversion Map

relacionadas ao banco de dados. Sendo assim, para atender a esta demanda, torna-se essencial o conhecimento, mesmo que não aprofundado, das nuances que envolvem tais atividades. Dentre elas, as questões relacionadas à estrutura da base de dados, tais como criação, alteração e manutenção, podem ser estabelecidas como essenciais ao processo de desenvolvimento. Diante disso, o presente artigo teve como foco o TMS Data Modeler, uma ferramenta comercial voltada à manipulação dos principais SGBDs disponíveis no cenário, e que estão inseridos no contexto Delphi, tais como Firebird, MS SQL Server, Oracle e MySQL.

Por conseguinte, esta é apenas uma, dentre as várias opções de ferramentas do gênero disponíveis no mercado. Logo, a intenção principal deste artigo foi a de expor os detalhes do produto que podem pesar a seu favor durante a escolha da ferramenta ideal a determinado ambiente/cotidiano de trabalho.

Autor

Fabrício Hissao Kawata
fabricio.kawata@bol.com.br
Formado em Processamento de Dados pela FATEC-TQ e pós-graduado em Engenharia de Componentes. Atua como Analista Programador Delphi há 9 anos. Atualmente escreve artigos sobre Delphi para revistas especializadas. Certified Delphi Developer. Certified Embarcadero Instructor. Site pessoal: www.fabriciokawata.com.br

Você gostou deste artigo?

Dê seu voto em ***www.devmedia.com.br/clubedelphi/feedback***

Ajude-nos a manter a qualidade da revista!

Somos tão apaixonados por tecnologia que o nome da empresa

diz tudo.

Porta 80 é o melhor que a Internet pode oferecer para sua empresa.

Já completamos 8 anos e estamos a caminho dos 80, junto com nossos clientes.

Adoramos tecnologia.
Somos uma equipe composta de gente que entende e gosta do que faz,

assim como você.

Estrutura

100% NACIONAL. Servidores de primeira linha, links de alta capacidade.

Suporte diferenciado

Treinamos nossa equipe para fazer mais e melhor. Muito além do esperado.

Serviços

Oferecemos a tecnologia mais moderna, serviços diferenciados e antenados com as suas necessidades.

1-to-1

Conhecemos nossos clientes. Atendemos cada necessidade de forma única.
Conheça!

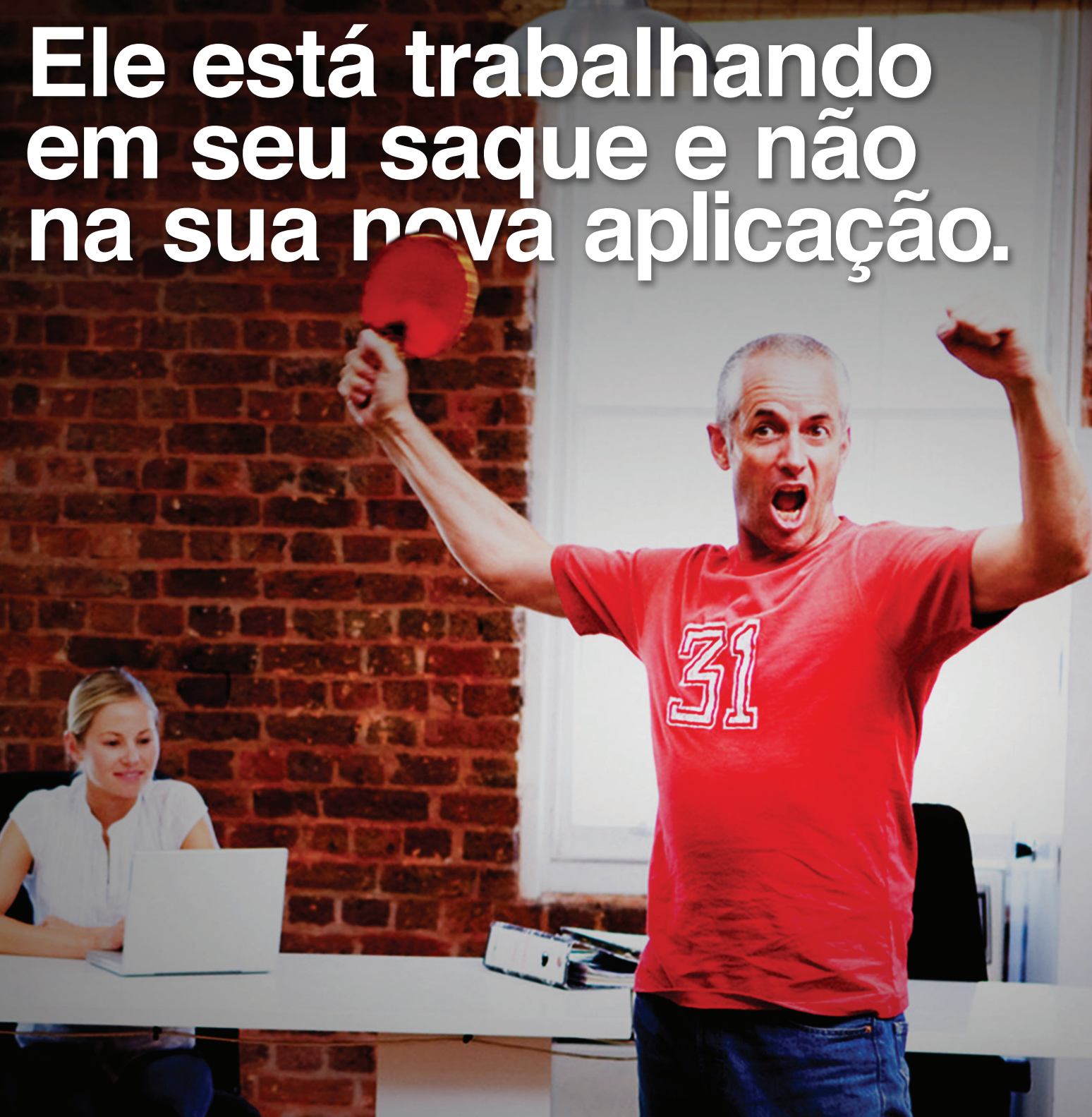
Porta 80
WEB HOSTING

Hospedagem | Cloud Computing | Dedicados | VoIP | Ecommerce | Aplicações | Streaming | Email corporativo

porta80.com.br | comercial@porta80.com.br | twitter.com/porta80

SP 4063-8616 | RJ 4063-5092 | MG 4063-8120 | DF 4063-7486

Ele está trabalhando em seu saque e não na sua nova aplicação.



Mais de 95%* dos times de desenvolvimento e testes de aplicativos relatam tempos de espera e atrasos para acessar os sistemas que necessitam para realizar suas atividades. A Virtualização de Serviços elimina estas dependências gerando ambientes simulados similares à realidade, permitindo o desenvolvimento em paralelo. Isto significa que suas aplicações serão lançadas mais rapidamente, com maior qualidade e menor custo. Game over!

Conheça mais em ca.com/br/GoDevOps

*De acordo com o estudo 2012 North America/Europe Service Virtualization Study, Coleman-Parkes.

ca technologies